Assignment 3 - Program Structures and Algorithms Spring 2023(SEC - 1)

NAME: Paurush Batish **NUID: 002755631**

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.

```
public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> fun
ction, 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
}
```

Solution –

```
 */
public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> function, UnaryOperator<T> preFunction, Consumer<U> postFunction) {
    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);
      }
      final double result = meanLapTime();
      return result;

}
private static long getClock() {
    // FIXME by replacing the following code
    return System.nanoTime();
    // END
}

/**
 * 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) {
    // FIXME by replacing the following code
     return ticks / 1_000_000.0;
    // END
}
```

Unit Test –

Finished after 1.529 seconds

Runs: 2/2 Errors: 0 Failures: 0

edu.neu.coe.info6205.util.BenchmarkTest [Runner: JUnit 4] (1.475 s)
- testWaitPeriods (1.474 s)
- getWarmupRuns (0.001 s)

Finished after 2.706 seconds

Runs: 11/11 Errors: 0 Failures: 0

edu.neu.coe.info6205.util.TimerTest [Runner: JUnit 4] (2.622 s)
- testPauseAndLapResume0 (0.192 s)
- testPauseAndLapResume1 (0.318 s)
- testLap (0.208 s)
- testPause (0.212 s)
- testStop (0.107 s)
- testMillisecs (0.106 s)
- testRepeat1 (0.128 s)
- testRepeat2 (0.248 s)
- testRepeat3 (0.624 s)
- testRepeat4 (0.375 s)
- testPauseAndLap (0.102 s)

Task - 2:

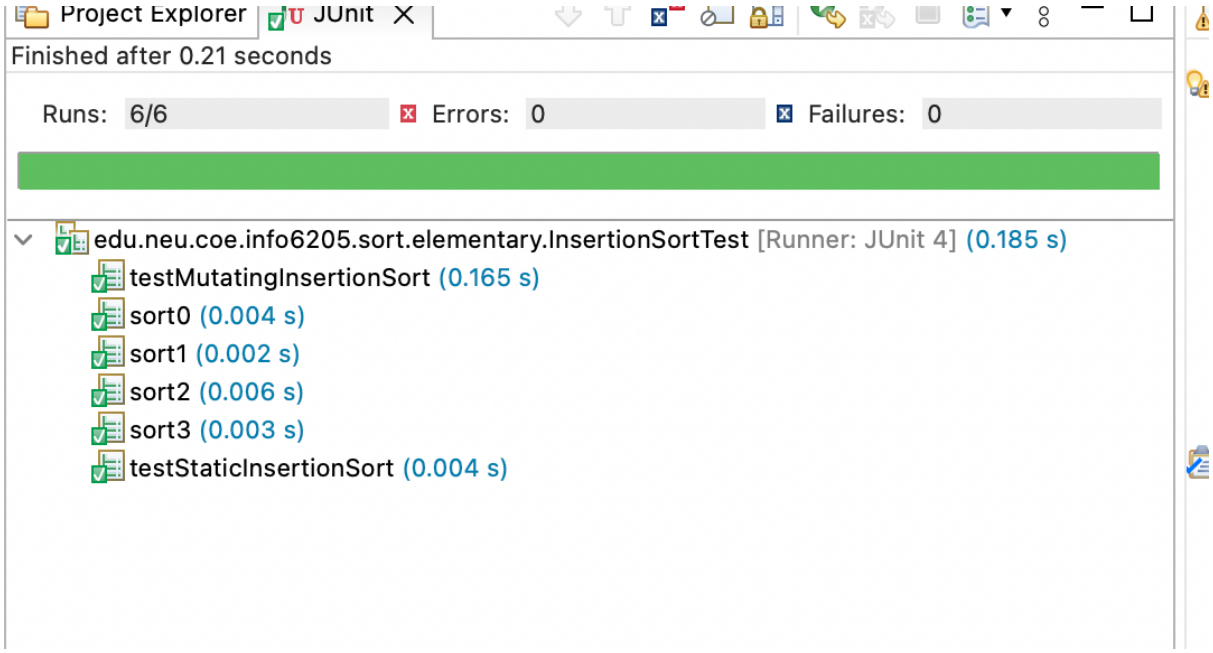
(Part 2) Implement *InsertionSort* (in the *InsertionSort* class) by simply looking up the insertion code used by *Arrays.sort*. If you have the *instrument = true* setting in *test/resources/config.ini*, then you will need to use the *helper* methods for comparing and swapping (so that they properly 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*.

```java
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-1;
        X curr=xs[i];
        while(j>=from && helper.compare(xs[j], curr)>0) {
            xs[j+1]=xs[j];
            j--;
            helper.incrementFixes(1);
        }
        xs[j+1]=curr;
    }

}
```

I have also implemented the insertion sort basic code –

```java
private void swap(int i, Object[] a) {
    // FIXME

    for (int j = i; j > 0; j--) {
        if (Integer.parseInt(a[j].toString()) > Integer.parseInt(a[j-1].toString())) {
            swap(a, j, j - 1);
        } else {
            break;
        }
    }
    // END
}
```

Unit Test –



Task - 3:

(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.

Solution –

Taking no of operations as 5

```
Random array:
Time elapsed: 23ms
Ordered array:
Time elapsed: 1ms
Partially ordered array:
Time elapsed: 17ms
Reverse ordered array:
Time elapsed: 7ms

Random array:
Time elapsed: 5ms
Ordered array:
Time elapsed: 1ms
Partially ordered array:
Time elapsed: 4ms
Reverse ordered array:
Time elapsed: 10ms

Random array:
Time elapsed: 19ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 19ms
Reverse ordered array:
Time elapsed: 38ms

Random array:
Time elapsed: 75ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 75ms
Reverse ordered array:
Time elapsed: 154ms
```

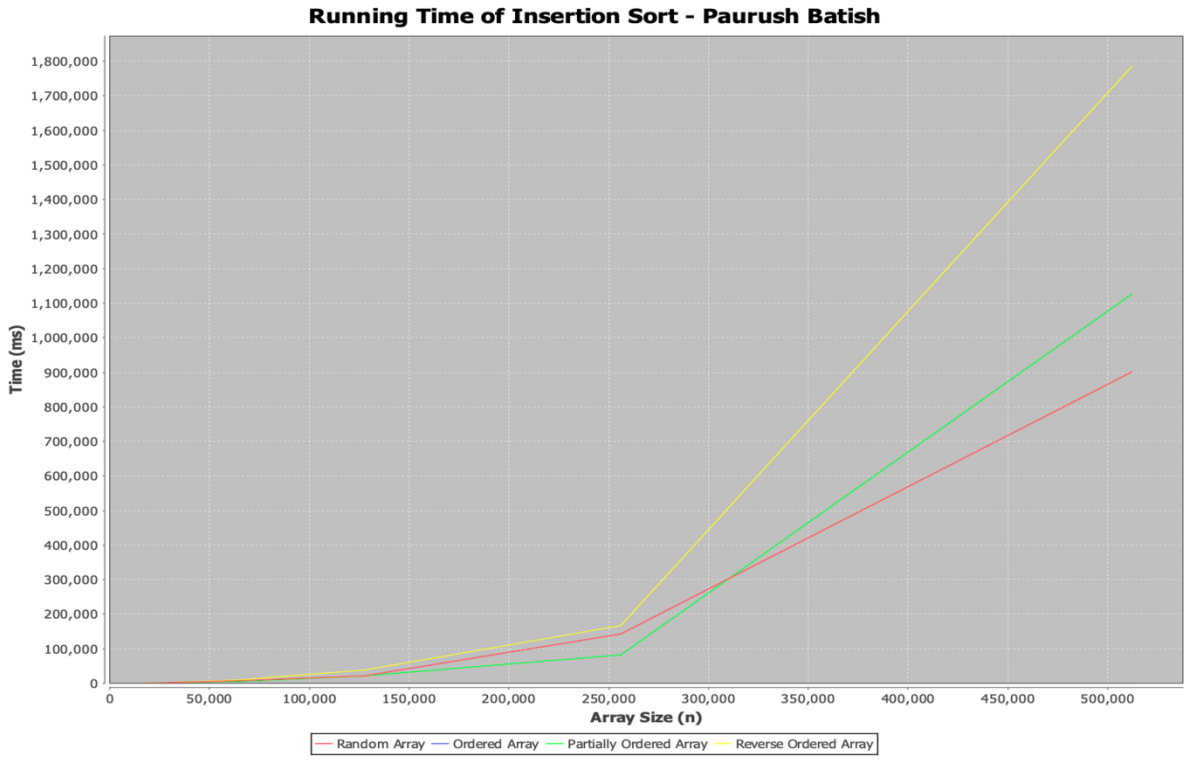
```
Random array:
Time elapsed: 369ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 304ms
Reverse ordered array:
Time elapsed: 607ms
```

Taking no of operations as 10

```
Random array:
Time elapsed: 27ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 17ms
Reverse ordered array:
Time elapsed: 6ms

Random array:
Time elapsed: 6ms
Ordered array:
```

```
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 5ms
Reverse ordered array:
Time elapsed: 9ms

Random array:
Time elapsed: 19ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 19ms
Reverse ordered array:
Time elapsed: 38ms

Random array:
Time elapsed: 75ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 77ms
Reverse ordered array:
Time elapsed: 153ms

Random array:
Time elapsed: 359ms
Ordered array:
Time elapsed: 1ms
Partially ordered array:
Time elapsed: 305ms
Reverse ordered array:
Time elapsed: 613ms

Random array:
Time elapsed: 1227ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 1220ms
Reverse ordered array:
Time elapsed: 2439ms

Random array:
Time elapsed: 7941ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 4877ms
Reverse ordered array:
```

```
Time elapsed: 9869ms

Random array:
Time elapsed: 22153ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 20949ms
Reverse ordered array:
Time elapsed: 39393ms

Random array:
Time elapsed: 141614ms
Ordered array:
Time elapsed: 1ms
Partially ordered array:
Time elapsed: 83242ms
Reverse ordered array:
Time elapsed: 167183ms

Random array:
Time elapsed: 902150ms
Ordered array:
Time elapsed: 2ms
Partially ordered array:
Time elapsed: 1126381ms
Reverse ordered array:
Time elapsed: 1784869ms
```



Based on the operations we can see that sorted array takes minimum time to perform insertion sort whereas reverse ordered array takes maximum time.

For a sorted list →

Since the list is already in sorted order the complexity will be in O(n).

For reverse sorted list →

We have to traverse n-1 times the inner loop and same with outer loop. Then the O(n*(n-1)) = O(n^2)