Program Structures and Algorithms
Spring 2023(SEC –)

NAME: Kasavaraju Venkata Pranay Kumar
NUID: 002701155

Task:

1. Implement three methods namely, repeat, getClock and toMilliSec according to business logic and requirement provided in assignment description.
2. Implement Insertion Sort by applying the algorithm by utilizing helper methods provided in the code.
3. Test the sorting algorithm implemented in part 2 under various scenarios such as using Random array, Ordered array , Partial-Ordered Array and Reverse-Ordered array as inputs. Use Doubling method to compute values and draw conclusions based on observations

Relationship Conclusion:
When analysing the performance of the insertion sort algorithm, it was observed that the input array's order had a significant impact on the algorithm's time complexity. Here are the conclusions that were drawn:

- Arrays that are already ordered resulted in the shortest processing time, with a performance that closely resembles a constant-time function.
- Partially ordered arrays took longer to process than ordered arrays, and the processing time increased at an exponential rate as the size of the array increased. This performance is similar to that of a $n(\log n)$^2 function, with a shallower slope compared to the random array case.
- Random arrays took even longer to process, with processing time increasing at a rate similar to that of $n(\log n)$^2, but with a steeper slope compared to the partially ordered arrays.
- Reverse ordered arrays took the longest to process, with processing time increasing at an exponential rate similar to $2n(\log n)$^2, and with the steepest slope among all the cases observed.

These findings highlight the importance of the order of the input array in determining the performance of the insertion sort algorithm.

Evidence to support that conclusion:
Observations n vs time for insertion sort:

N	Random (nanosec)	Ordered (nanosec)	Reverse Ordered (nanosec)	Partial Ordered (nanosec)
10	1045	34	4281	221
20	2211	76	3158	309
40	2738	147	3997	1327
80	10411	304	15335	7045
160	34098	567	60955	25269

```
=======================Random Ordered Array=======================
========================================================================
Test for n = 10
Time taken: 1045.0 Runs: 10000
========================================================================
Test for n = 20
Time taken: 4211.0 Runs: 10000
========================================================================
Test for n = 40
Time taken: 2738.0 Runs: 10000
========================================================================
Test for n = 80
Time taken: 10411.0 Runs: 10000
========================================================================
Test for n = 160
Time taken: 34098.0 Runs: 10000
```

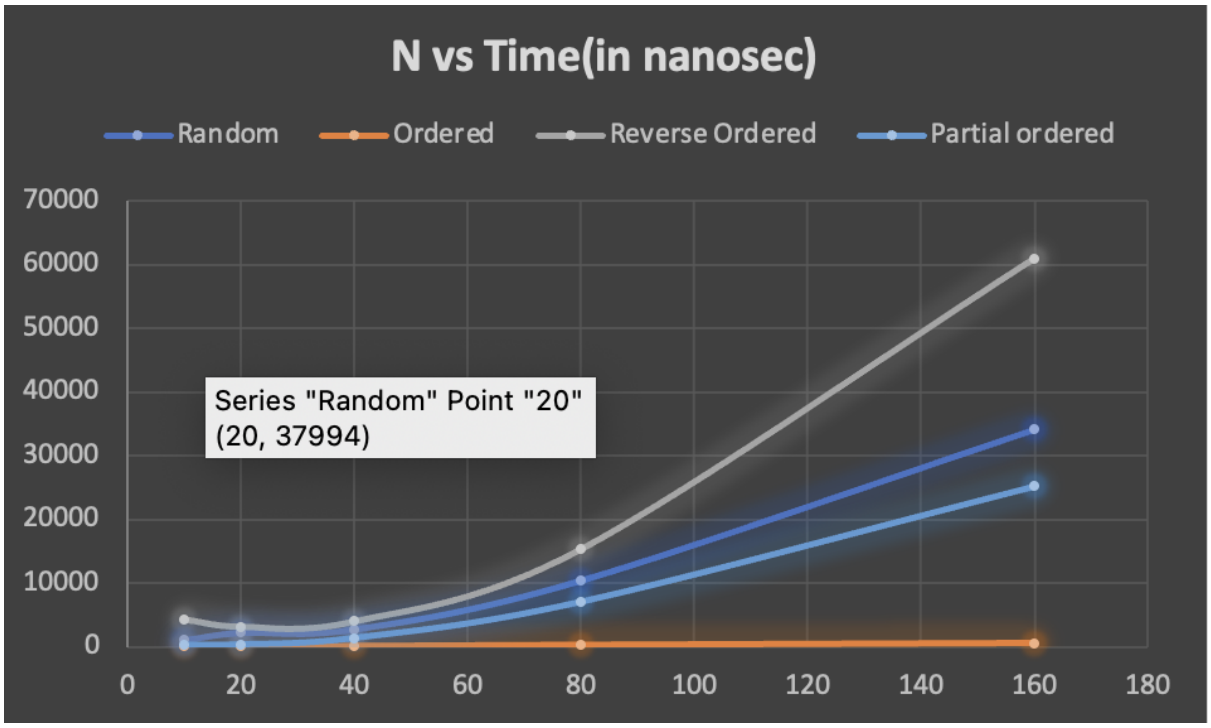
```
=======================Partial Ordered Array=======================
========================================================================
Test for n = 10
Time taken: 221.0 Runs: 10000
========================================================================
Test for n = 20
Time taken: 309.0 Runs: 10000
========================================================================
Test for n = 40
Time taken: 1327.0 Runs: 10000
========================================================================
Test for n = 80
Time taken: 7045.0 Runs: 10000
========================================================================
Test for n = 160
Time taken: 25269.0 Runs: 10000
```

```
=======================Reverse Ordered Array=======================
========================================================================
Test for n = 10
Time taken: 4281.0 Runs: 10000
========================================================================
Test for n = 20
Time taken: 3158.0 Runs: 10000
========================================================================
Test for n = 40
Time taken: 3997.0 Runs: 10000
========================================================================
Test for n = 80
Time taken: 15335.0 Runs: 10000
========================================================================
Test for n = 160
Time taken: 60955.0 Runs: 10000
```

```
=======================Ordered Array=======================
========================================================================
Test for n = 10
Time taken: 34.0 Runs: 10000
========================================================================
Test for n = 20
Time taken: 76.0 Runs: 10000
========================================================================
Test for n = 40
Time taken: 147.0 Runs: 10000
========================================================================
Test for n = 80
Time taken: 304.0 Runs: 10000
========================================================================
Test for n = 160
Time taken: 567.0 Runs: 10000
```

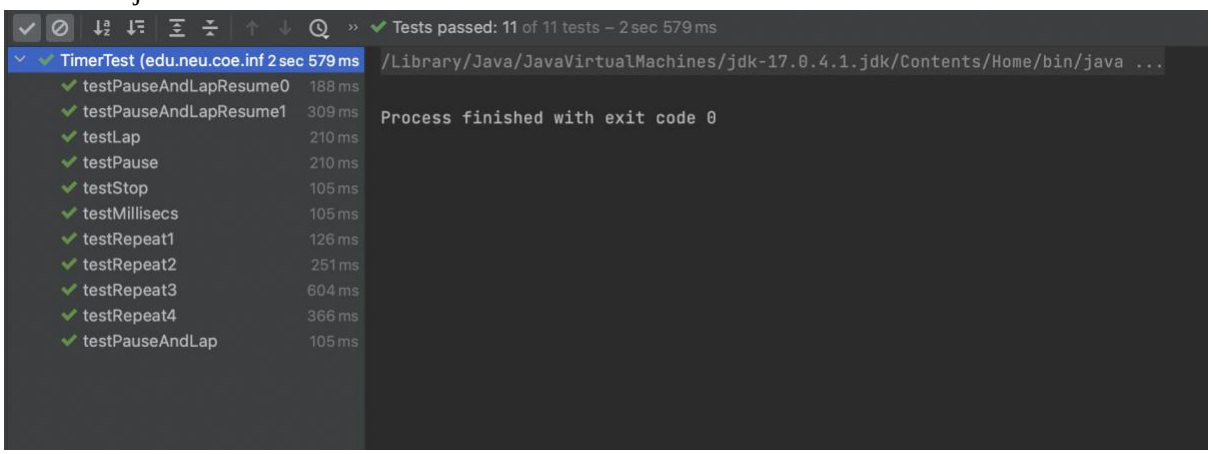
Graphical Representation:

1. N Vs Time



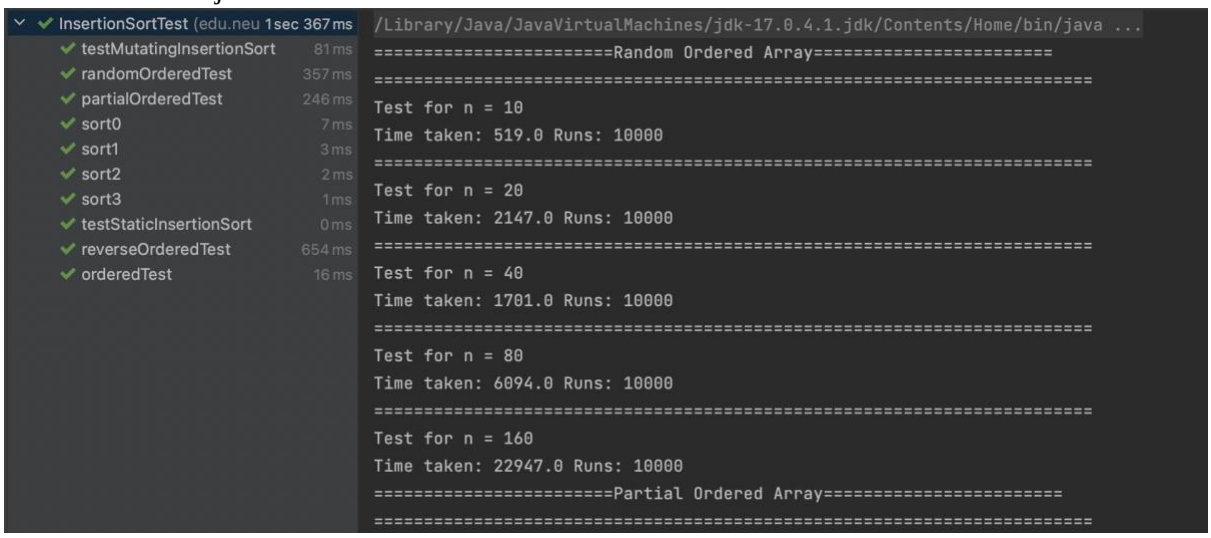
Unit Test Screenshots:

TimerTest.java



BenchmarkTest.java

InsertionSortTest.java



Code Snippets:

Part 1:

Timer.java: repeat()

```java
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");

    if(running){pause();}

    for (int i = 0; i < n; i++) {
        T value = supplier.get();
        if (preFunction != null) {
            value = preFunction.apply(value);
        }
        resume();
        U result = function.apply(value);
        pauseAndLap();
        if (postFunction != null) {
            postFunction.accept(result);
        }
    }
    final double result = meanLapTime();
    resume();
    return result;
    // END
}
```

Timer.java: getClock() and toMillisecs()

```java
private static long getClock() {
    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.
 */
2 usages  Kasavaraju-v *
private static double toMillisecs(long ticks) {
    return (double)ticks / 1e+6;
    // END
}
```

Part 2:
InsertionSort.java:

```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;
        while (j >= from) {
            if (helper.less(xs[j + 1], xs[j])) {
                helper.swap(xs, j, j + 1);
                j--;
            } else {
                break;
            }
        }
    }
    // END
}
```

Part-3: InsertionSortTest.java: code for all four types of input arrays.

```java
@Test
public void randomOrderedTest() throws Exception{
    System.out.println("=======================Random Ordered Array=======================");
    int[] values = {10, 20, 40, 80, 160};
    for(int n: values) {
        System.out.println("================================================================================");
        System.out.println("Test for n = " + n);
        Integer[] inputArray = generateRandomArray(n);
        int runs = 10000;

        double time = benchmarkRunTime(runs, n, inputArray);

        System.out.println("Time taken: " + time + " Runs: " + runs);
    }
}

new *
@Test
public void orderedTest() throws Exception{
    System.out.println("=======================Ordered Array=======================");
    int[] values = {10, 20, 40, 80, 160};
    for(int n: values) {
        System.out.println("================================================================================");
        System.out.println("Test for n = " + n);
        Integer[] inputArray = generateRandomArray(n);
        Arrays.sort(inputArray);

        int runs = 10000;

        double time = benchmarkRunTime(runs, n, inputArray);

        System.out.println("Time taken: " + time + " Runs: " + runs);
    }
}
```

```java
@Test
public void reverseOrderedTest() throws Exception{
    System.out.println("=======================Reverse Ordered Array=======================");
    int[] values = {10, 20, 40, 80, 160};
    for(int n: values) {
        System.out.println("================================================================================");
        System.out.println("Test for n = " + n);
        Integer[] inputArray = generateRandomArray(n);
        Arrays.sort(inputArray, Collections.reverseOrder());

        int runs = 10000;

        double time = benchmarkRunTime(runs, n, inputArray);

        System.out.println("Time taken: " + time + " Runs: " + runs);
    }
}

new *
@Test
public void partialOrderedTest() throws Exception{
    System.out.println("=======================Partial Ordered Array=======================");
    int[] values = {10, 20, 40, 80, 160};
    for(int n: values) {
        System.out.println("================================================================================");
        System.out.println("Test for n = " + n);
        Integer[] inputArray = generateRandomArray(n);
        Arrays.sort(inputArray, fromIndex: 0, toIndex: (2*n)/5);
        int runs = 10000;

        double time = benchmarkRunTime(runs, n, inputArray);

        System.out.println("Time taken: " + time + " Runs: " + runs);
    }
}
```

Helper functions

```
    private Integer[] generateRandomArray(int n) {
        Random random = new Random();
        Integer[] randomIntegers = new Integer[n];
        for(int i=0; i<n; i++){
            randomIntegers[i] = random.nextInt(n);
        }
        return randomIntegers;
    }

    4 usages  new *
    private double benchmarkRunTime(int runs, int n, Integer[] inputArray){
        Integer[] temp = Arrays.copyOf(inputArray, n);

        long time = 0;
        InsertionSort insertionSort = new InsertionSort();
        for(int i=0; i<runs; i++){

            long startTime = System.nanoTime();
            insertionSort.sort(temp,  from: 0, n);
            long endTime = System.nanoTime();

            time += (endTime - startTime);

            temp = Arrays.copyOf(inputArray, n);
        }
        return time/runs;
    }

    final static LazyLogger logger = new LazyLogger(InsertionSort.class);

}
```