Program structures and algorithms
Spring 2023 (Section-01)

BY: PRANAV KAPOOR – NUID: 002998253

Assignment 3 (Benchmark)

(Part 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> 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*. If you have trouble with the exact timings in the unit tests, it's quite OK (in this assignment only) to change parameters until the tests run. Different machine architectures will result in different behaviour.

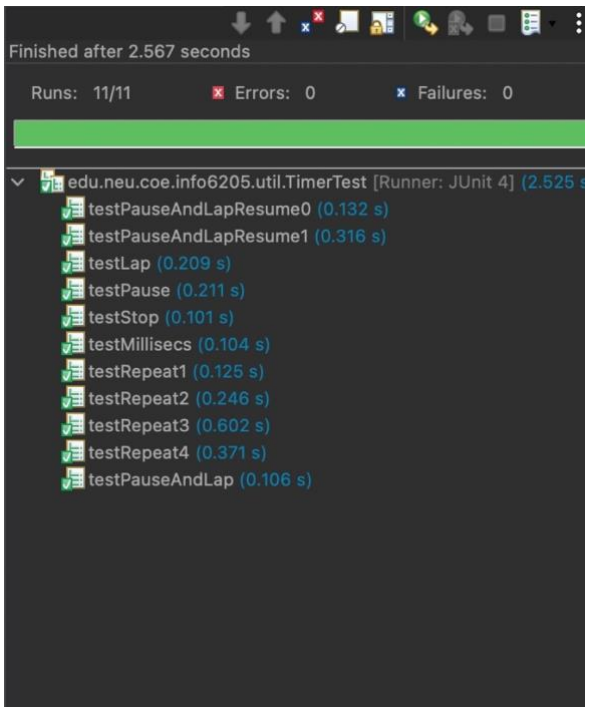
Solution:

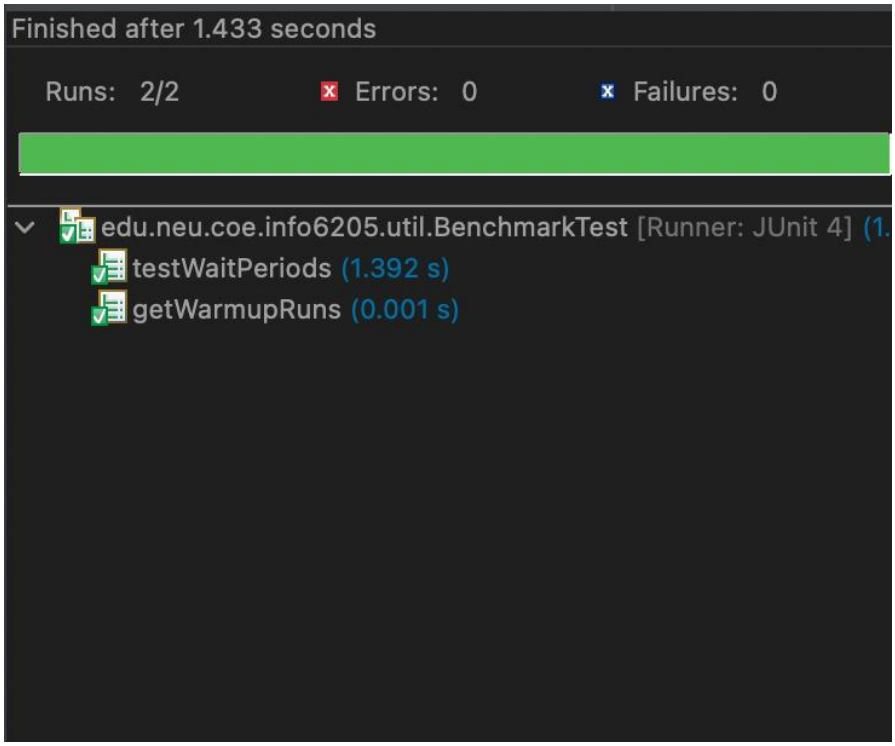
```
public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> function, UnaryOperator<T> preFunction, Consumer<U> postFunction) {
    logger.trace("repeat: with " + n + " runs");
    // FIXME: note that the timer is running when this method is called and should still be running when it returns. by replacing the following code
    pause(); // pause the timer

    T pre = supplier.get();
    while(n!=0) {
        if (preFunction != null)

            pre = preFunction.apply(supplier.get()); // getting the supply not timed
        resume(); // start the timer again

        U u = function.apply(pre); // this will be timed !!!

        pauseAndLap(); //pause the timer postFunction !!
        if (postFunction != null)
            postFunction.accept(u);

        n--;
    }
    final double result = meanLapTime();
    return result;
    // END
}
```

```
     * @return the corresponding number of milliseconds.
     */
    private static double toMillisecs(long ticks) {
        // FIXME by replacing the following code
        return ticks / 1000000.0;
//      return 0;
        // END
    }
```

```
     */
    private static long getClock() {
        // FIXME by replacing the following code
        return System.nanoTime();
//      return 0;
        // END
    }

```

Unit Tests





(Part 2)

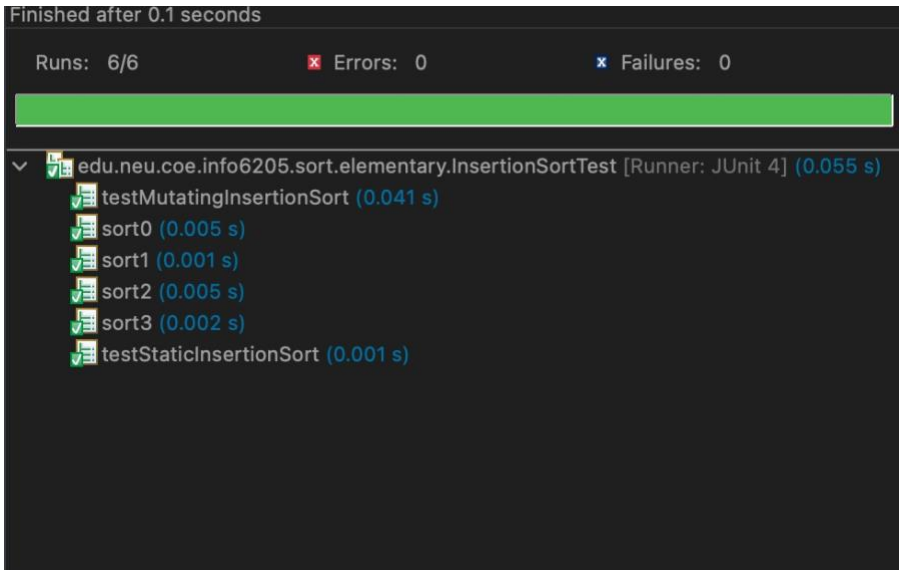
Implement *InsertionSort* (in the *InsertionSort* class) by simply looking up the insertion code used by *Arrays.sort*. If you have the *instrument = true* setting in *test/resources/config.ini*, then you will need to use the *helper* methods for comparing and swapping (so that they properly count the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

Solution

```
    */
    public void sort(X[] xs, int start, int end) {

        //FIXME
        final Helper<X> helper = getHelper();
        for (int i = start + 1; i < end; i++) {
            int j = i;
            while (j > start && helper.swapStableConditional(xs, j)) {
                j--;
            }
        }
        //END
    }
```

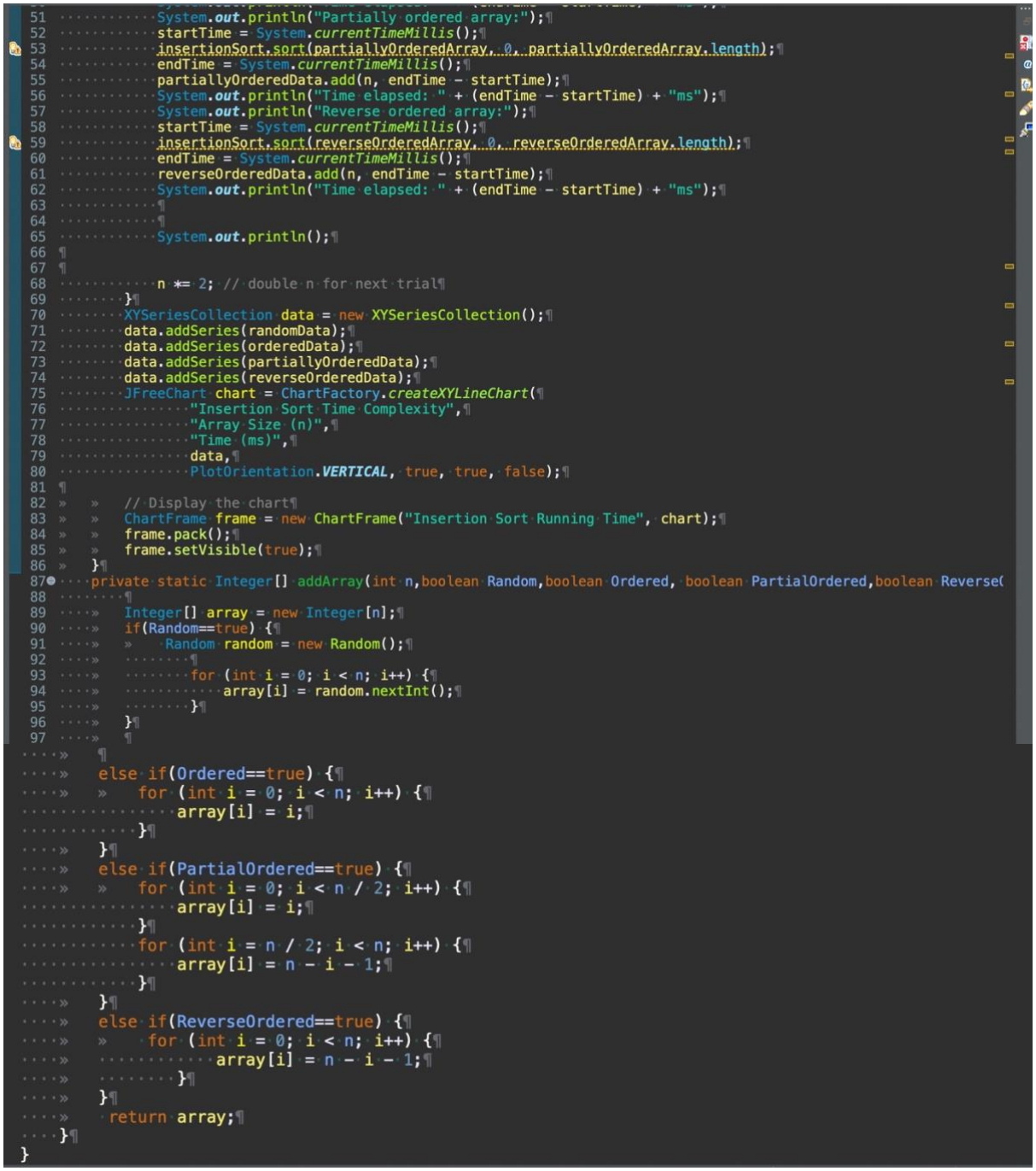
Unit Test



(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.

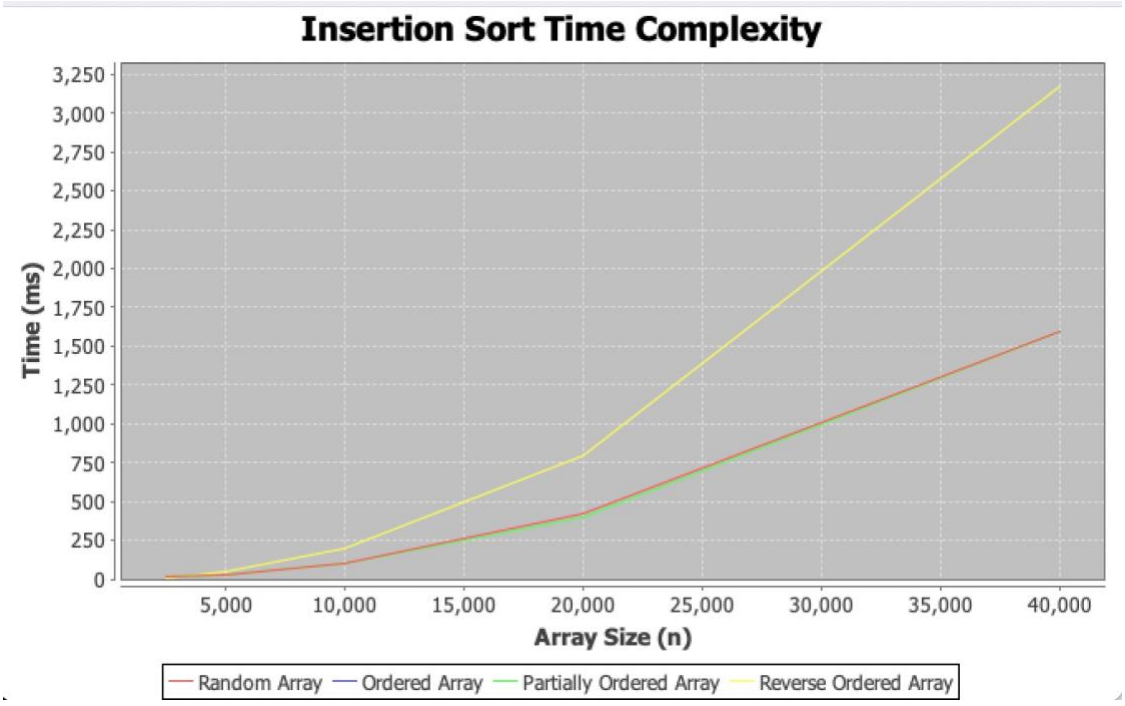
```
package edu.neu.coe.info6205.sort.elementary;
import java.util.Random;


import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartFrame;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.jfree.ui.ApplicationFrame;


public class mainMethodApproach extends ApplicationFrame {
    public mainMethodApproach(String title) {
        super(title);
    }

    public static void main(String[] args) {
        
        mainMethodApproach mainProgram = new mainMethodApproach("Insertion Sort Time Complexity");
        Integer[] randomArray, orderedArray, partiallyOrderedArray, reverseOrderedArray;
        InsertionSort insertionSort = new InsertionSort();

        int n = 3000; // starting value for n
        int numTrials = 10; // number of trials for each value of n

        XYSeries randomData = new XYSeries("Random Array");
        XYSeries orderedData = new XYSeries("Ordered Array");
        XYSeries partiallyOrderedData = new XYSeries("Partially Ordered Array");
        XYSeries reverseOrderedData = new XYSeries("Reverse Ordered Array");

        for (int i = 0; i < numTrials; i++) {
            randomArray = addArray(n,true,false,false,false);
            orderedArray = addArray(n,false,true,false,false);
            partiallyOrderedArray = addArray(n,false,false,true,false);
            reverseOrderedArray = addArray(n,false,false,false,true);
            System.out.println("Random array:");
            long startTime = System.currentTimeMillis();
            insertionSort.sort(randomArray, 0, randomArray.length);
            long endTime = System.currentTimeMillis();
            randomData.add(n, endTime - startTime);
            System.out.println("Time elapsed: " + (endTime - startTime) + "ms");
            System.out.println("Ordered array:");
            startTime = System.currentTimeMillis();
            insertionSort.sort(orderedArray, 0, orderedArray.length);
            endTime = System.currentTimeMillis();
            orderedData.add(n, endTime - startTime);
            System.out.println("Time elapsed: " + (endTime - startTime) + "ms");
```



```
            System.out.println("Partially ordered array:");
            startTime = System.currentTimeMillis();
            insertionSort.sort(partiallyOrderedArray, 0, partiallyOrderedArray.length);
            endTime = System.currentTimeMillis();
            partiallyOrderedData.add(n, endTime - startTime);
            System.out.println("Time elapsed: " + (endTime - startTime) + "ms");
            System.out.println("Reverse ordered array:");
            startTime = System.currentTimeMillis();
            insertionSort.sort(reverseOrderedArray, 0, reverseOrderedArray.length);
            endTime = System.currentTimeMillis();
            reverseOrderedData.add(n, endTime - startTime);
            System.out.println("Time elapsed: " + (endTime - startTime) + "ms");


            System.out.println();


            n *= 2; // double n for next trial
        }
        XYSeriesCollection data = new XYSeriesCollection();
        data.addSeries(randomData);
        data.addSeries(orderedData);
        data.addSeries(partiallyOrderedData);
        data.addSeries(reverseOrderedData);
        JFreeChart chart = ChartFactory.createXYLineChart(
                "Insertion Sort Time Complexity",
                "Array Size (n)",
                "Time (ms)",
                data,
                PlotOrientation.VERTICAL, true, true, false);

        // Display the chart
        ChartFrame frame = new ChartFrame("Insertion Sort Running Time", chart);
        frame.pack();
        frame.setVisible(true);
    }
    private static Integer[] addArray(int n,boolean Random,boolean Ordered, boolean PartialOrdered,boolean Reverse(

        Integer[] array = new Integer[n];
        if(Random==true) {
            Random random = new Random();

            for (int i = 0; i < n; i++) {
                array[i] = random.nextInt();
            }
        }


    else if(Ordered==true) {
        for (int i = 0; i < n; i++) {
            array[i] = i;
        }
    }
    else if(PartialOrdered==true) {
        for (int i = 0; i < n / 2; i++) {
            array[i] = i;
        }
        for (int i = n / 2; i < n; i++) {
            array[i] = n - i - 1;
        }
    }
    else if(ReverseOrdered==true) {
        for (int i = 0; i < n; i++) {
            array[i] = n - i - 1;
        }
    }
    return array;
}
}
```

Unit Test output
N=2500, no of trials=5

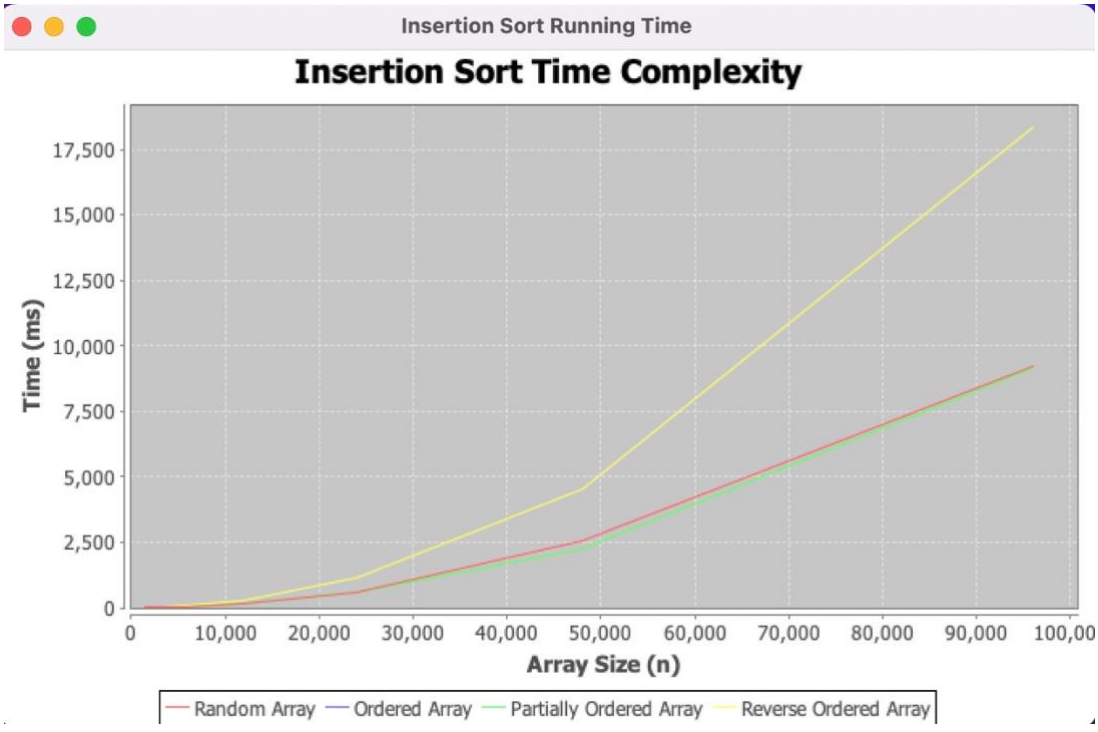
```
mainMethodApproach [Java Application] /Users/Pranavkapoor1/Library/Java/JavaVirtualMachines/openjdk-17.0.2/Contents/Home/bin/java  (04-Feb-20
Random array:
Time elapsed: 19ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 13ms
Reverse ordered array:
Time elapsed: 12ms

Random array:
Time elapsed: 25ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 25ms
Reverse ordered array:
Time elapsed: 49ms

Random array:
Time elapsed: 99ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 99ms
Reverse ordered array:
Time elapsed: 200ms

Random array:
Time elapsed: 422ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 399ms
Reverse ordered array:
Time elapsed: 797ms

Random array:
Time elapsed: 1609ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 1598ms
Reverse ordered array:
Time elapsed: 3162ms
```



Conclusion: Looking at the graph, it is evident that Insertion sort for the Reverse ordered array has the highest increasing time complexity, followed by Random array, partially ordered array, and then ordered array.

N=1500, no of t srials=7

```
mainMethodApproach [Java Application] /Users/Pranavkapoor1/Library/Java/JavaVirtualMachines/openjdk-17.0.2/Contents/Home/bin/java  (04-Feb-2023, 12:05:53 pm)
Random array:
Time elapsed: 23ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 7ms
Reverse ordered array:
Time elapsed: 16ms

Random array:
Time elapsed: 9ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 9ms
Reverse ordered array:
Time elapsed: 18ms

Random array:
Time elapsed: 36ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 36ms
Reverse ordered array:
Time elapsed: 71ms

Random array:
Time elapsed: 144ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 144ms
Reverse ordered array:
Time elapsed: 286ms

Random array:
Time elapsed: 577ms
Ordered array:
Time elapsed: 0ms
Partially ordered array:
Time elapsed: 575ms
Reverse ordered array:
Time elapsed: 1148ms

Random array:
Time elapsed: 2549ms
Ordered array:
Time elapsed: 1ms
Partially ordered array:
Time elapsed: 2286ms
Reverse ordered array:
Time elapsed: 4563ms

Random array:
Time elapsed: 9201ms
Ordered array:
```



Conclusion: Looking at the graph, it is evident that Insertion sort for the Reverse ordered array has the highest increasing time complexity, followed by Random array, partially ordered array, and then ordered array.