Student Name: **Hongyao Tao (001067209)**

INFO 6205
Program Structures & Algorithms
Spring 2021

Assignment No.2

- **The implement of three methods of Timer class and test result**

```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");
    // TO BE IMPLEMENTED: note that the timer is running when this method is called and should still be running when it returns.
    running=false;
    ticks=0;

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

```java
/**
 * Get the number of ticks from the system clock.
 * <p>
 * NOTE: (Maintain consistency) There are two system methods for
 * Ensure that this method is consistent with toMillisecs.
 *
 * @return the number of ticks for the system clock. Currently d
 */
private static long getClock() { return System.nanoTime(); }
```

```java
/**
 * NOTE: (Maintain consistency) There are two system methods for getting the clock t
 * 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) { return ticks / Math.pow(10, 6); }
```
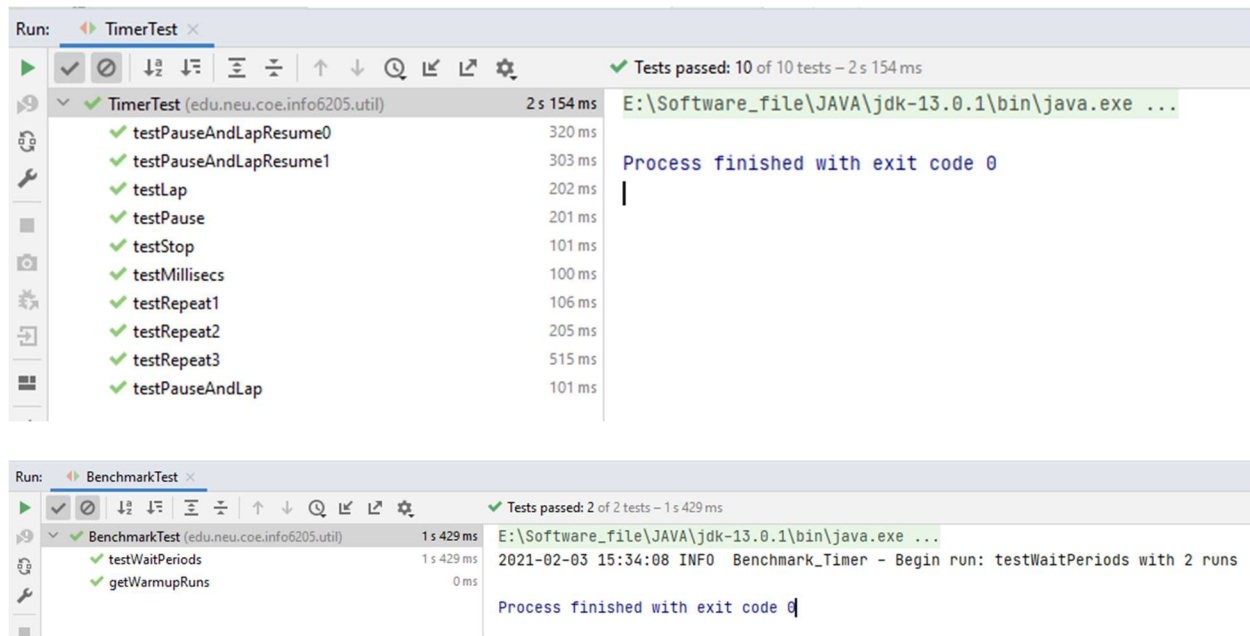
Figure 1. The test result of Timer class

- **The implement of insertionSort class and test result**

```java
/**
 * Sort the sub-array xs:from:to using insertion sort.
 *
 * @param xs    sort the array xs from "from" to "to".
 * @param from  the index of the first element to sort
 * @param to    the index of the first element not to sort
 */
public void sort(X[] xs, int from, int to) {
    final Helper<X> helper = getHelper();
    for (int start = from; start < to - 1; start++) {
        for (int sortIndex = start; sortIndex >= 0; sortIndex--) {
            int nextIndex = sortIndex + 1;
            if (helper.compare(xs, sortIndex, nextIndex) > 0) {
                helper.swap(xs, sortIndex, nextIndex);
            } else {
                break;
            }
        }
    }
}
```
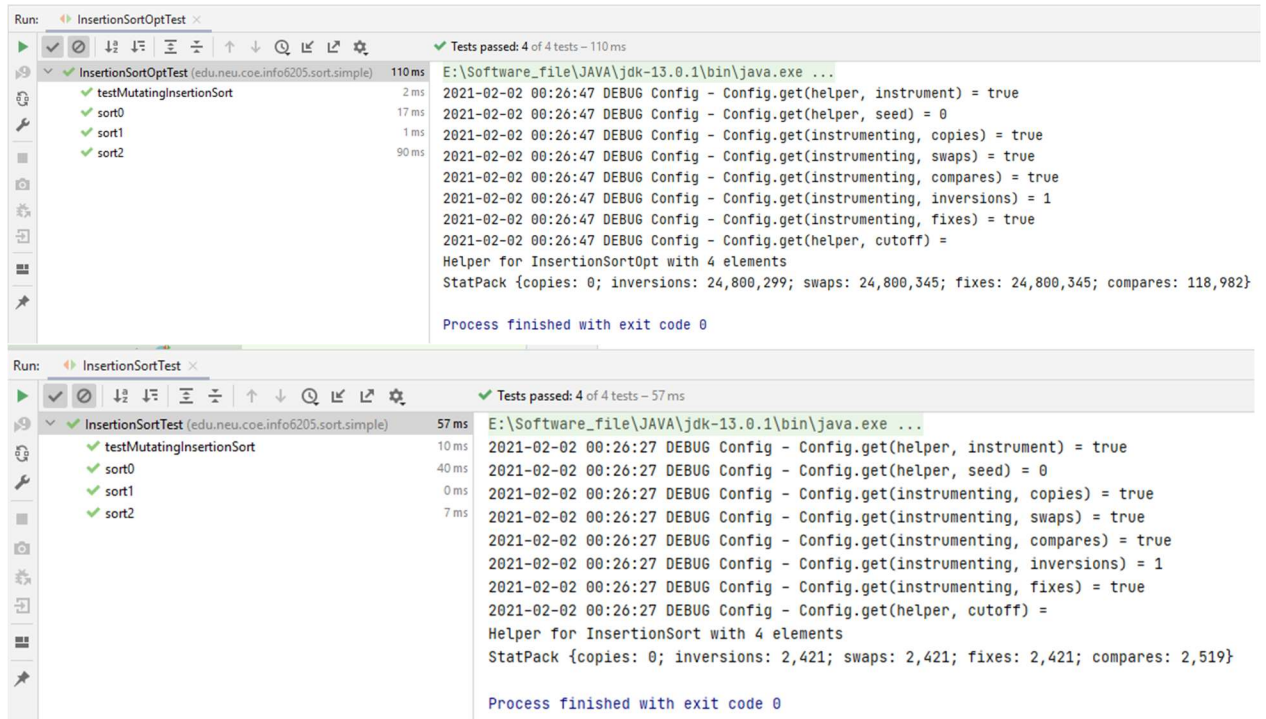
Figure 2. The Test result of insertion sort class

- **The implement of unit test for insertion sort benchmarks on different input data**

```java
public class BenchmarksInsertionSortTest {

    final static LazyLogger logger = new LazyLogger(Benchmarks.class);
    public static void writeToCSV(String fileName,String line){
        FileWriter fw = null;
        try {
            fw = new FileWriter(new File(fileName), append: true);
            fw.write( str: line+"\n");
            fw.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    @Test
    public void randomTest(){
        int initialN=500;
        writeToCSV( fileName: "randomSort.csv", line: "N,Time");
        for(int i=0;i<7;i++){
            initialN*=2;
            String description="Random generator";
            Helper<Integer> helper=new BaseHelper<>(description,initialN);
            InsertionSort<Integer> insertionSort= new InsertionSort<Integer>(helper);
            Supplier<Integer[]> supplier = () -> helper.random(Integer.class, r -> r.nextInt());
            Benchmark<Integer[]> benchmark = new Benchmark_Timer<>(
                    description: description + " for " + initialN + " Integers",
                    (xs) -> Arrays.copyOf(xs, xs.length),
                    insertionSort::mutatingSort,
                    fPost: null
            );
            double average=benchmark.runFromSupplier(supplier, m: 50);
            writeToCSV( fileName: "randomSort.csv", line: initialN+","+average);
            logger.info("Average milionSecond :"+average);
        }
    }
}
```

```java
@Test
public void orderedTest(){
    int initialN=500;
    writeToCSV( fileName: "orderedSort.csv", line: "N,Time");
    for(int i=0;i<7;i++){
        initialN*=2;
        String description="ordered generator";
        Helper<Integer> helper=new BaseHelper<>(description);
        InsertionSort<Integer> insertionSort= new InsertionSort<Integer>(helper);

        final int finalInitialN = initialN;
        Supplier<Integer[]> supplier = () -> {
            Integer[] data=new Integer[finalInitialN];
            for(int j=0;j<finalInitialN;j++){
                data[j]=j;
            }
            return data;
        };
        Benchmark<Integer[]> benchmark = new Benchmark_Timer<>(
                description: description + " for " + initialN + " Integers",
                (xs) -> Arrays.copyOf(xs, xs.length),
                insertionSort::mutatingSort,
                fPost: null
        );
        double average=benchmark.runFromSupplier(supplier, m: 50);
        writeToCSV( fileName: "orderedSort.csv", line: initialN+","+average);
        logger.info("Average milionSecond :"+average);
    }
}
```

```java
@Test
public void particiallyOrderedTest(){
    int initialN=500;
    writeToCSV( fileName: "particialOrderedSort.csv", line: "N,Time");
    Random random=new Random();
    for(int i=0;i<7;i++){
        initialN*=2;
        String description="ordered generator";
        Helper<Integer> helper=new BaseHelper<>(description);
        InsertionSort<Integer> insertionSort= new InsertionSort<Integer>(helper);

        final int finalInitialN = initialN;
        Supplier<Integer[]> supplier = () -> {
            Integer[] data=new Integer[finalInitialN];
            for(int j=0;j<finalInitialN;j++){
                data[j]= random.nextInt(finalInitialN);
            }
            int orderCount= (int) (finalInitialN*0.3);
            int startOrdedIndex=random.nextInt( bound: finalInitialN-orderCount);
            for (int j=startOrdedIndex;j<finalInitialN;j++){
                data[j]=startOrdedIndex;
            }
            return data;
        };
        Benchmark<Integer[]> benchmark = new Benchmark_Timer<>(
                description: description + " for " + initialN + " Integers",
                (xs) -> Arrays.copyOf(xs, xs.length),
                insertionSort::mutatingSort,
                fPost: null
        );
        double average=benchmark.runFromSupplier(supplier, m: 50);
        writeToCSV( fileName: "particialOrderedSort.csv", line: initialN+","+average);
        logger.info("Average milionSecond :"+average);
    }
}
```

```java
@Test
public void reverseOrderedTest(){
    int initialN=500;
    writeToCSV( fileName: "reverseSort.csv", line: "N,Time");
    for(int i=0;i<7;i++){
        initialN*=2;
        String description="reverse generator";
        Helper<Integer> helper=new BaseHelper<>(description);
        InsertionSort<Integer> insertionSort= new InsertionSort<Integer>(helper);

        final int finalInitialN = initialN;
        Supplier<Integer[]> supplier = () -> {
            Integer[] data=new Integer[finalInitialN];
            for(int j=0;j<finalInitialN;j++){
                data[j]=finalInitialN-j;
            }
            return data;
        };
        Benchmark<Integer[]> benchmark = new Benchmark_Timer<>(
                description: description + " for " + initialN + " Integers",
                (xs) -> Arrays.copyOf(xs, xs.length),
                insertionSort::mutatingSort,
                fPost: null
        );
        double average=benchmark.runFromSupplier(supplier, m: 50);
        writeToCSV( fileName: "reverseSort.csv", line: initialN+","+average);
        logger.info("Average milionSecond :"+average);
    }
}
```

- **Conclusion**

N	Time		N	Time		N	Time		N	Time
1000	0.00374		1000	0.926396		1000	1.30684		1000	2.490344
2000	0.007288		2000	2.997794		2000	5.247338		2000	9.869888
4000	0.013932		4000	12.87756		4000	20.04287		4000	39.47166
8000	0.028252		8000	44.14877		8000	83.71686		8000	159.5778
16000	0.058246		16000	196.8301		16000	341.8605		16000	641.8089
32000	0.107032		32000	808.0965		32000	1522.587		32000	2595.685
64000	0.212534		64000	3579.477		64000	6969.604		64000	10740.45

Figure 3. The T/N grow result on ordered, partial ordered, random and reversed input
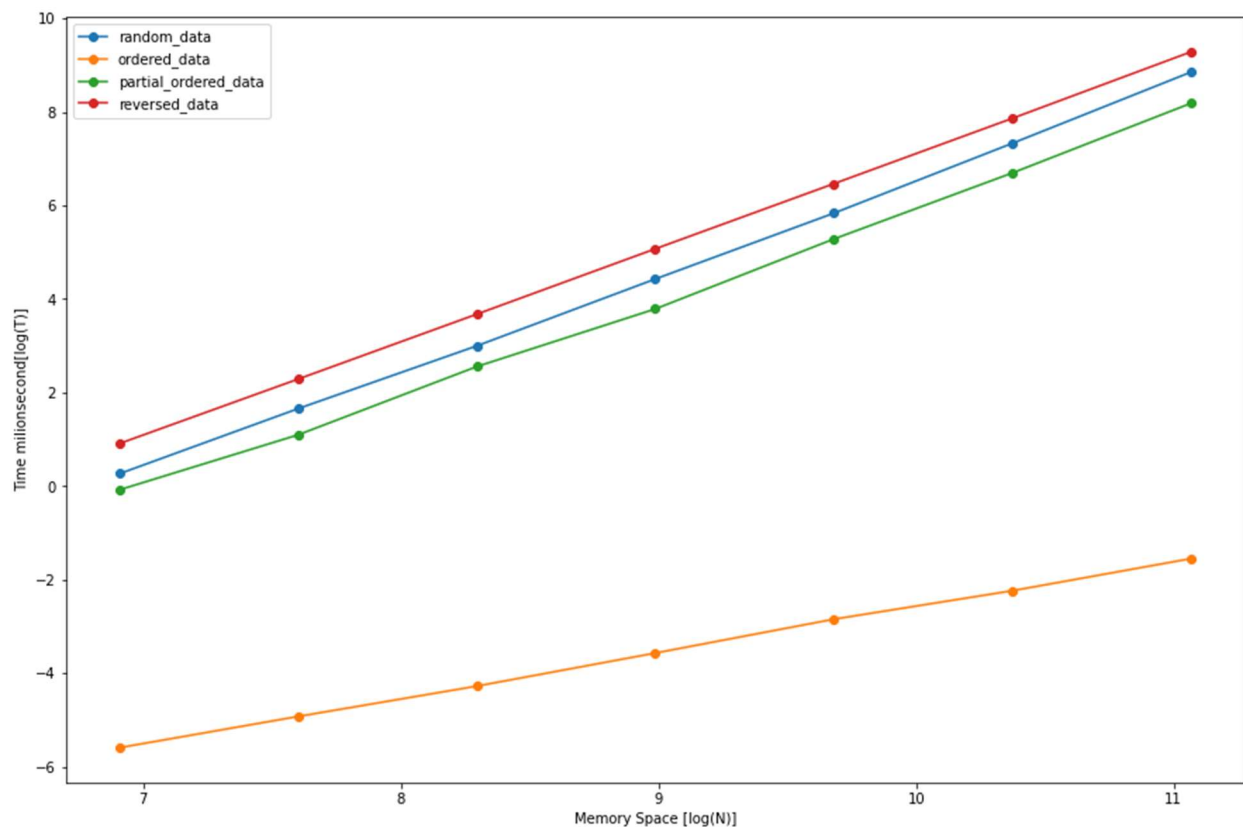

Figure 4. The insertion sort T/N grow logarithm graph

According the insertion sort logarithm graph, when the input data is reversed, the cost of time is biggest which is more than the cost of time on random data. Besides, time cost of sorting

random data is a bit more than sorting partial ordered data. When the input data is ordered, this sorting algorithm has the minimal time cost which is far less three previous situations. In addition, this sorting algorithm has liner increasement on this logarithm graph, which means that the subtraction of log(T2) and log(T1) is constant that is log(T2)-log(T1) ~ log(2N*(2N+1)/2)-log(N*(N+1)/2) ~ log 4.