Assignment 3

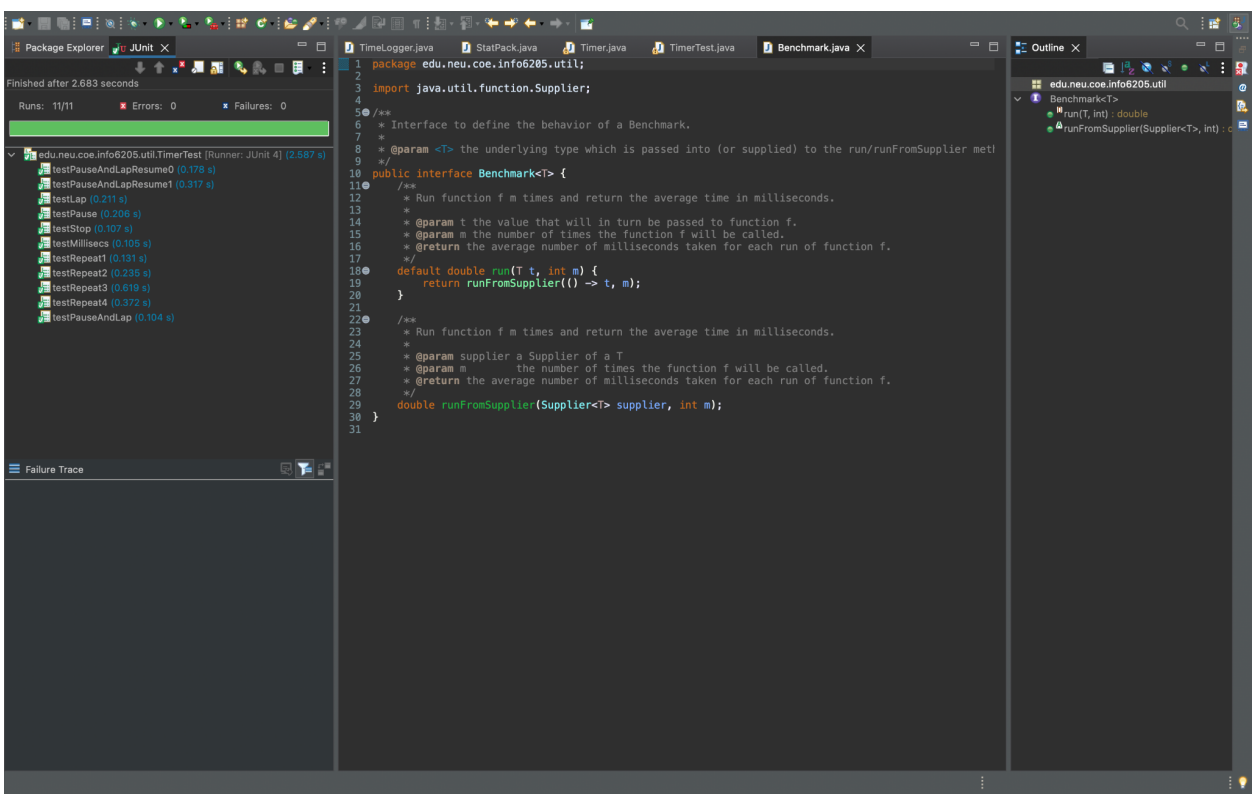
Name : Chandan Anandachari

NUID: 002777668

Your task for this assignment is in three parts.

- (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. The APIs of these class are as follows:

Answer : Implemented the repeat getClock and toMillisecs of the Timer class



Package Explorer | JUnit

Finished after 1.524 seconds

Runs: 2/2 Errors: 0 Failures: 0

edu.neu.coe.info6205.util.BenchmarkTest [Runner: JUnit 4] (1.41)
- testWaitPeriods (1.415 s)
- getWarmupRuns (0.000 s)

Failure Trace

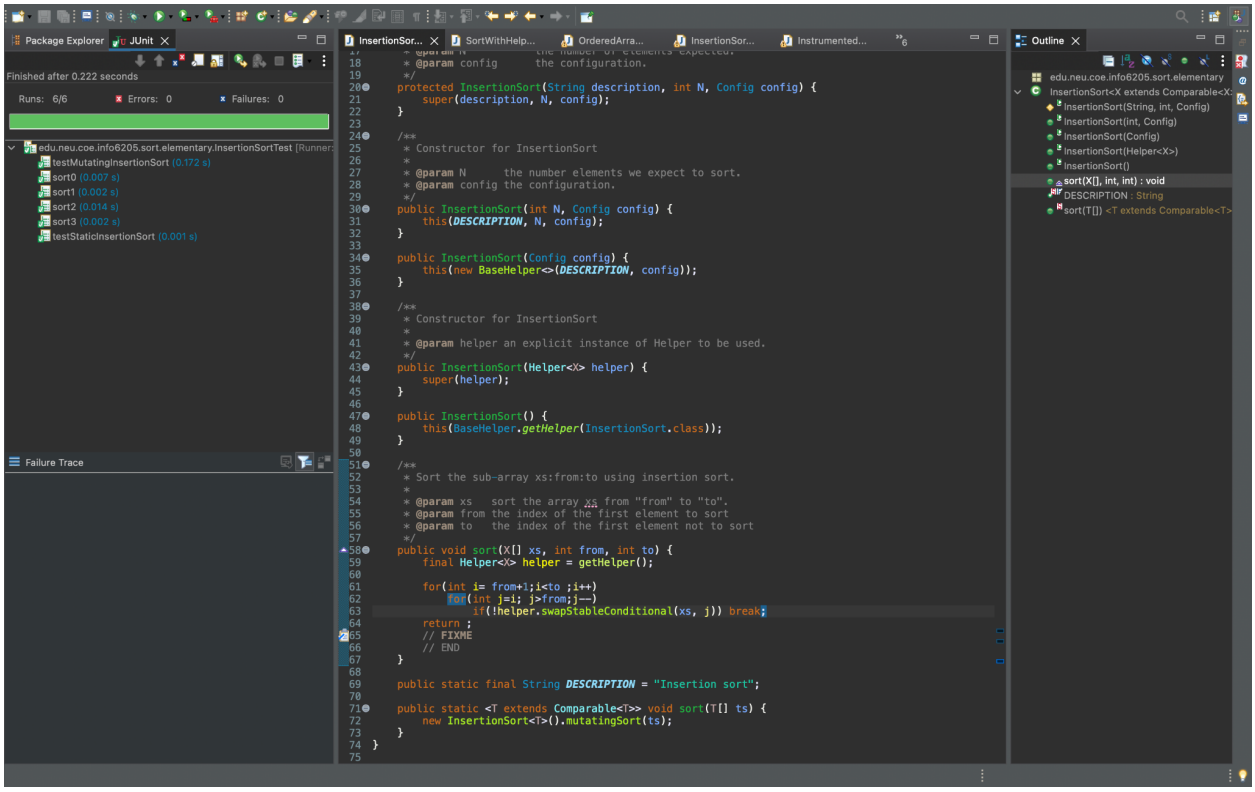
TimeLogger.java | StatPack.java | Timer.java | TimerTest.java | Benchmark.java

```java
package edu.neu.coe.info6205.util;

import java.util.function.Supplier;

/**
 * Interface to define the behavior of a Benchmark.
 *
 * @param <T> the underlying type which is passed into (or supplied) to the run/runFromSupplier met
 */
public interface Benchmark<T> {
    /**
     * Run function f m times and return the average time in milliseconds.
     *
     * @param t the value that will in turn be passed to function f.
     * @param m the number of times the function f will be called.
     * @return the average number of milliseconds taken for each run of function f.
     */
    default double run(T t, int m) {
        return runFromSupplier(() -> t, m);
    }

    /**
     * Run function f m times and return the average time in milliseconds.
     *
     * @param supplier a Supplier of a T
     * @param m        the number of times the function f will be called.
     * @return the average number of milliseconds taken for each run of function f.
     */
    double runFromSupplier(Supplier<T> supplier, int m);
}
```

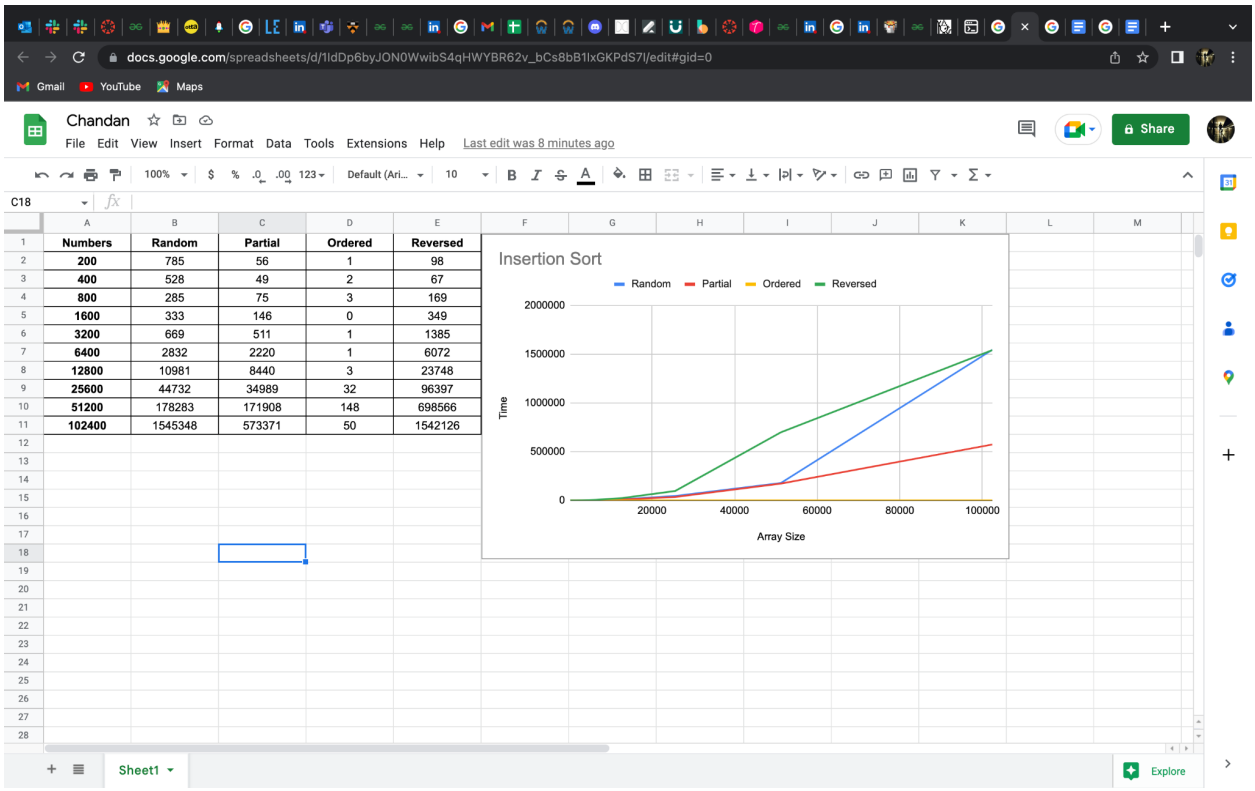
Outline

- edu.neu.coe.info6205.util
  - Benchmark<T>
    - run(T, int) : double
    - runFromSupplier(Supplier<T>, int) : c

- (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*.



- (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.



Numbers	Random	Partial	Ordered	Reversed
200	785	56	1	98
400	528	49	2	67
800	285	75	3	169
1600	333	146	0	349
3200	669	511	1	1385
6400	2832	2220	1	6072
12800	10981	8440	3	23748
25600	44732	34989	32	96397
51200	178283	171908	148	698566
102400	1545348	573371	50	1542126

Random Array : all the values are picked randomly using the built-in random function.

Partial Array : the partial array where half of the array is filled with random numbers using the random built-in function. And the reset of the array is arranged in sorted way.

Ordered Array : where the array is sorted in ascending order.

Reversed Array : where the array is sorted in descending order for a given value of array.

Observation ;

From the above data and the time taken by the insertion sort to sort all the different kinds of values in array the best case is when the array is sorted and it takes the least time and the second array takes best case is with the partial sorted array where it almost takes the constant time and it has the log n time, where the ordered take almost like constant time. While the partial and reversed sorted arrays takes the worst time. Therefore from the above observation i can conclude that insertion sort takes less time to sort the sorted array where as it take to much time for the reversed array to be sorted. Because i has to make to many swaps and comparisons.