

Venkata Sai Akhil Vemula (002981289)

Program Structures & Algorithms Fall 2021 Assignment No. 2

A) Tasks Performed:

1) Implemented Methods in Timer.java

```
/**
 * Pause (without counting a lap); run the given functions n times while bei
 *
 * @param n the number of repetitions.
 * @param supplier a function which supplies a T value.
 * @param function a function T->U and which is to be timed.
 * @param preFunction a function which pre-processes a T value and which
 * @param postFunction a function which consumes a U and which succeeds the
 * @return the average milliseconds per repetition.
 */
public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> fu
    logger.trace("repeat: with " + n + " runs");
    // TO BE IMPLEMENTED; note that the timer is running when this method
    return 0;
}

/**
 * Stop this Timer and return the mean lap time in milliseconds.
 *
 * @return the average milliseconds used by each lap.
 * @throws TimerException if this Timer is not running.
 */
public double stop() {
    pauseAndLap();
    return meanLapTime();
}

/**
 * Return the mean lap time in milliseconds for this paused timer.
 *
 * @return the average milliseconds used by each lap.
 */
private double meanLapTime() {
    // TO BE IMPLEMENTED
    return 0;
}

/**
 * NOTE: (Maintain consistency) There are two system methods for getting 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) {
    // TO BE IMPLEMENTED
    return 0;
}

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

static class TimerException extends RuntimeException {
    public TimerException() {
    }

    public TimerException(String message) {
        super(message);
    }
}

/**
 * Pause (without counting a lap); run the given functions n times while bei
 *
 * @param n the number of repetitions.
 * @param supplier a function which supplies a T value.
 * @param function a function T->U and which is to be timed.
 * @param preFunction a function which pre-processes a T value and which
 * @param postFunction a function which consumes a U and which succeeds the
 * @return the average milliseconds per repetition.
 */
public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> func
    logger.trace("repeat: with " + n + " runs");
    pause();
    for (int i = 0; i < n; i++) {
        if (preFunction!=null)preFunction.apply(supplier.get());
        resume();
        U t = function.apply(supplier.get());
        pauseAndLap();
        if (postFunction!=null) postFunction.accept(t);
    }
    return meanLapTime();
}

/**
 * Stop this Timer and return the mean lap time in milliseconds.
 *
 * @return the average milliseconds used by each lap.
 * @throws TimerException if this Timer is not running.
 */
public double stop() {
    pauseAndLap();
    return meanLapTime();
}

/**
 * Return the mean lap time in milliseconds for this paused timer.
 *
 * @return the average milliseconds used by each lap.
 */
private double meanLapTime() {
    // TO BE IMPLEMENTED
    return 0;
}

/**
 * NOTE: (Maintain consistency) There are two system methods for getting the
 * 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 TimeUnit.NANOSECONDS.toMillis(ticks);
}

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

static class TimerException extends RuntimeException {
    public TimerException() {
    }

    public TimerException(String message) {
        super(message);
    }

    public TimerException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

2) Implemented Insertion Sort logic for both the cases (instrument = true and false)

```
public InsertionSort(Helpler<I> helpler) {
    super(helpler);
}

public InsertionSort() {
    this(BaseHelpler.getHelpler(InsertionSort.class));
}

/**
 * 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(I[] xs, int from, int to) {
    final Helpler<I> helpler = getHelpler();

    // TO BE IMPLEMENTED

    public static final String DESCRIPTION = "Insertion sort";

    public static <I extends Comparable<I>> void sort(I[] ts) {
        new InsertionSort<I>().mutatingSort(ts);
    }
}
```

```
/**
 * 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(I[] xs, int from, int to) {
    final Helpler<I> helpler = getHelpler();
    // if instrument = true; use helper methods else use own insertion sort swaps
    if (helpler.instrumented()) {
        for (int i = from; i < to-1; i++) {
            for (int j = i+1; j > from && helpler.swapStableConditional(xs, j); j--);
        }
    } else {
        for (int i = from; i < to-1; i++) {
            for (int j = i+1; j > from && xs[j].compareTo(xs[j-1]) < 0; j--) {
                I x = xs[j-1];
                xs[j-1] = xs[j];
                xs[j] = x;
            }
        }
    }

    public static final String DESCRIPTION = "Insertion sort";

    public static <I extends Comparable<I>> void sort(I[] ts) {
        new InsertionSort<I>().mutatingSort(ts);
    }
}
```

3) Added Main Method for running Tests on Insertion Sort. Also added logic to get output in the form of CSV.

```
public void sort(I[] xs, int from, int to) {
    final Helpler<I> helpler = getHelpler();
    // if instrument = true; use helper methods else use own insertion sort swaps
    if (helpler.instrumented()) {
        for (int i = from; i < to-1; i++) {
            for (int j = i+1; j > from && helpler.swapStableConditional(xs, j); j--);
        }
    } else {
        for (int i = from; i < to-1; i++) {
            for (int j = i+1; j > from && xs[j].compareTo(xs[j-1]) < 0; j--) {
                I x = xs[j-1];
                xs[j-1] = xs[j];
                xs[j] = x;
            }
        }
    }

    public static final String DESCRIPTION = "Insertion sort";

    public static <I extends Comparable<I>> void sort(I[] ts) {
        new InsertionSort<I>().mutatingSort(ts);
    }
}
```

```
public static void main(String[] args) {
    StringBuilder outputBuilder = new StringBuilder();
    outputBuilder.append("SL.NO,")
        .append("Array Length(n),")
        .append("Random Array,")
        .append("Partially Ordered Array,")
        .append("Ordered Array,")
        .append("Reverse Ordered Array")
        .append("\n");

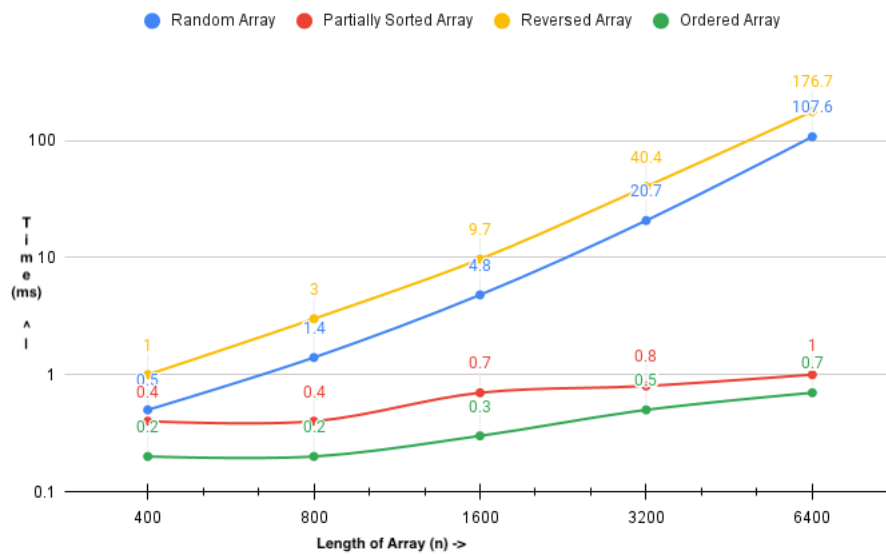
    Benchmark_Timer<Integer[]> benchmark_timer = new Benchmark_Timer<>() {
        "Insertion Sort Benchmark",
        numsArray -> new InsertionSort<Integer>().sort(numsArray, 0, numsArray.length);
    };

    int limit = 128000;
    for (int i = 1; i < 6; i++) {
        limit /= 2;
        outputBuilder.append(i).append(", ").append(limit).append(", ");
        final int bound = limit * 10;
        Integer[] numArray = IntStream.generate(() -> new Random().nextInt(bound))
            .limit(limit)
            .boxed()
            .toArray(Integer[]::new);

        double randomSortTime = benchmark_timer.runFromSupplier(numArray::clone, 10);
        outputBuilder.append(randomSortTime).append(", ");
        Arrays.sort(numArray, 0, numArray.length / 2);
        double partiallyOrderedTime = benchmark_timer.runFromSupplier(() -> numArray, 10);
        outputBuilder.append(partiallyOrderedTime).append(", ");
        Arrays.sort(numArray);
        double orderedSortTime = benchmark_timer.runFromSupplier(() -> numArray, 10);
        outputBuilder.append(orderedSortTime).append(", ");
        double reversedSortTime = benchmark_timer.runFromSupplier(() -> reverseArray(numArray), 10);
        outputBuilder.append(reversedSortTime).append("\n");
    }
}
```

B) Analysis

Graph 1: time (ms) vs length of array



- From the values obtained, it can be observed that it takes longer time to sort reversed array compared to others.
- It takes longer time to sort random array compared to Partially ordered and ordered arrays. Next comes partially ordered, it lies between random and ordered as shown in graph.
- It takes very less time for processing Ordered Array.

Google Sheet URL : <https://docs.google.com/spreadsheets/d/1zli1ynr85myV2n-GUD-QXE8twzSWI8NFFzbWE6uAF8/edit?usp=sharing>

C) Unit Tests & Main Code Output Screenshots

Image C1) BenchmarkTests

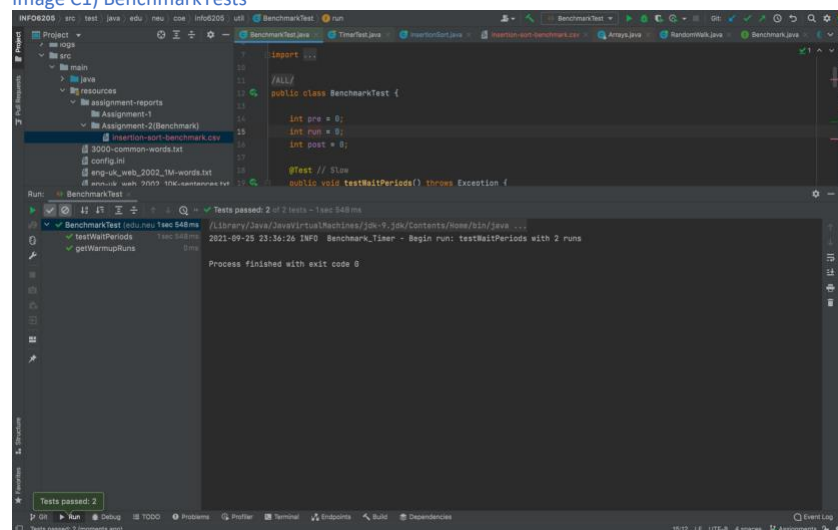


Image C2) Timer Tests

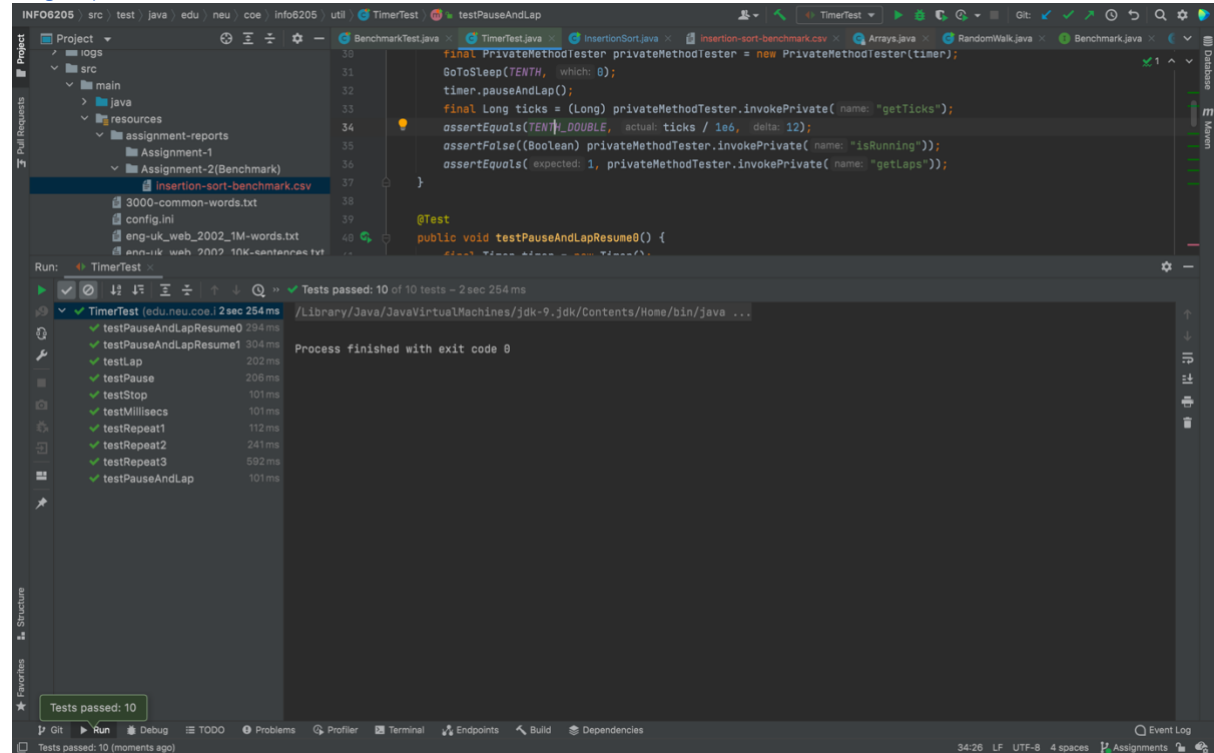


Image C3) InsertionSortTests

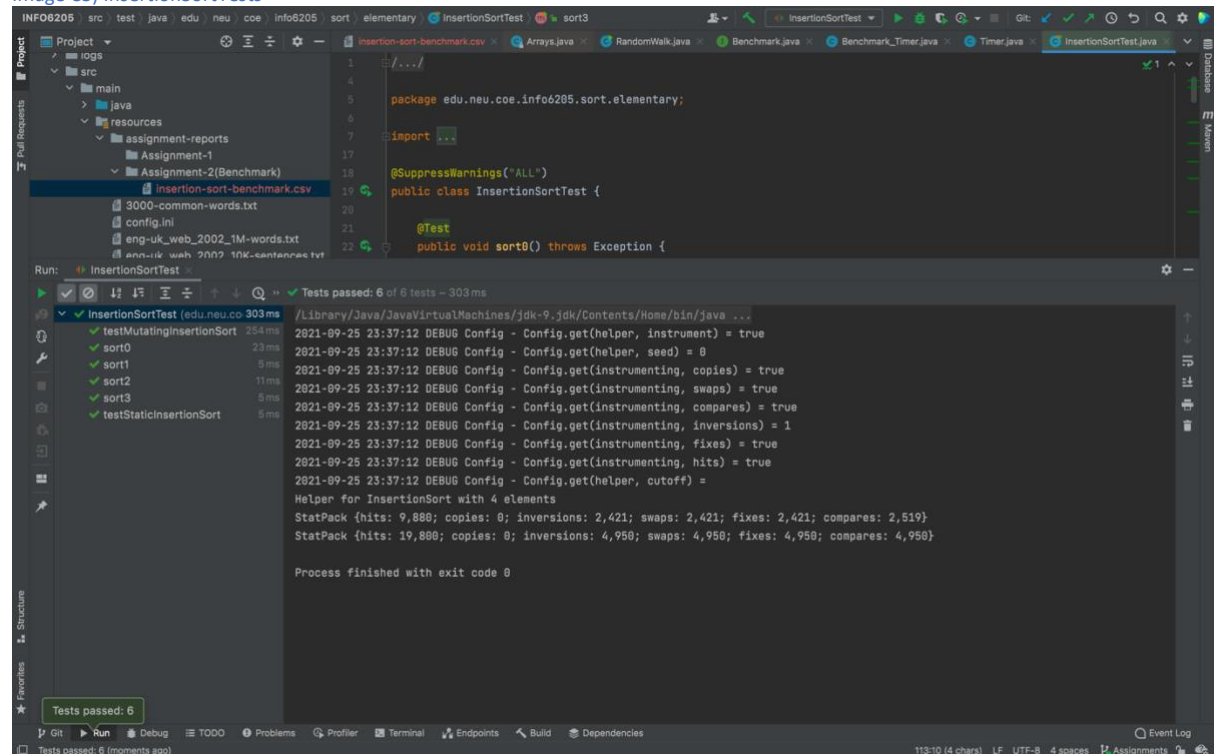


Image C4) Main Method Output

