Assignment 3 (Benchmark)

1. Intro

In this question, I was asked to finish three tasks. There are

> - Implement three (3) methods (*repeat*, *getClock*, and *toMillisecs*) of a class called *Timer*.
> - Implement *InsertionSort* (in the *InsertionSort* class).
> - Implement a main program to measure the running times of four different arrays: random, ordered, partially-ordered and reverse-ordered.

2. Implement

2.1 Timer

```
    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");
        pause();
        for (int i = 0; i < n; i++) {
            T preVal = supplier.get();
            if (Objects.nonNull(preFunction)){
                preVal = preFunction.apply(preVal);
            }
            resume();
            //Only calculate the time spent in the process of fRun
            U runVal =  function.apply(preVal);
            pauseAndLap();
```

```
            if(Objects.nonNull(postFunction)){
                postFunction.accept(runVal);
            }
        }
        double time = meanLapTime();
        resume();
        return time;
        // END
    }
```

When we want to implement this method, we need to figure out how this method works.

> - **supplier:** When you want to test a sort, you can use supplier to generate a random array.
> - **preFunction:** Each time when we want to sort, we need a random array, but it's too time-consuming to generate random arrays every time. At this time, we can use preFunction to copy the random array generated by the supplier and use it.
> - **function:** Run **Sort()**.
> - **postFunction:** The processed business logic does not participate in timing.

To make the timing more accurate, we only calculate the time of the running function. Therefore, every time we run the function. We need to reset time and pause the timer when it fininsh.

```
            resume();
            //Only calculate the time spent in the process of fRun
            U runVal =  function.apply(preVal);
            pauseAndLap();
```

2.2 Insertion Sort

```
    /**
     * 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 i = from + 1; i < to; i++) {
            int j = i;
            while (j > from && helper.less(xs[i], xs[j - 1])) {
                j--;
            }
            helper.swapInto(xs, j, i);
        }
        // END
    }
```

We only need to use the knowledge learned in the previous lesson to complete the insertion sort.

2.3 Benchmarks

I wrote a unit test in **InsertionSortTest.class** and I just need to pass four different initial array into repeat method.

Besides, I used supplier to generate four different types of arrays.

```
    public static final String RANDOM_ARRAY = "Random Array";
    public static final String ORDERED_ARRAY = "Ordered Array";
```

```java
    public static final String PARTIALLY_ORDERED_ARRAY = "Partially Ordered
Array";
    public static final String REVERSE_ORDERED_ARRAY = "Reverse Ordered
Array";
/**
 * Generate four types of arr
 *
 * @param type initial arr type
 * @param len    length
 * @return supplier
 */
public static Supplier<Integer[]> initialArr(String type, int len) {
    switch (type) {
        case RANDOM_ARRAY:
            return getSupplier(0, len, 0, 0, len);
        case ORDERED_ARRAY:
            return getSupplier(0, 0, 0, len, len);
        case PARTIALLY_ORDERED_ARRAY:
            return getSupplier(0, len / 2, len / 2, len, len);
        case REVERSE_ORDERED_ARRAY:
            return () -> {
                Integer[] result = (Integer[])
Array.newInstance(Integer.class, len);
                for (int i = len - 1; i >= 0; i--) result[i] = len - 1 - i;
                return result;
            };
    }
    return null;
}

    /**
     * @param rStart generate random arr from rStart to rEnd
     * @param rEnd   end index
     * @param sStart generate sorted arr from sStart to sEnd
```

```
     * @param sEnd   end index
     * @param length arr length
     * @return supplier
     */
    private static Supplier<Integer[]> getSupplier(int rStart, int rEnd,
int sStart, int sEnd, int length) {
        final Random random = new Random();
        final Supplier<Integer[]> integersSupplier = () -> {
            Integer[] result = (Integer[]) Array.newInstance(Integer.class,
length);
            for (int i = rStart; i < rEnd; i++) result[i] =
random.nextInt();
            for (int i = sStart; i < sEnd; i++) result[i] = i;
            return result;
        };
        return integersSupplier;
    }

    public static double runBenchmarkTimer(Supplier<Integer[]> supplier,
int m){
        return new Benchmark_Timer<Integer[]>(
                "insertionSort",
                (xs) -> Arrays.copyOf(xs, xs.length),
                InsertionSort::sort,
                null
        ).runFromSupplier(supplier, m);
    }
```

3. Test

I used 6 various lengh of arrays to run the test. Each time, every method will run 100 times. The length of array will start at 1000, and it will be doubled by each loop.

```
/**
 * In this method, measure the running times of insertion sort, using four
different initial array ordering
 * situations: random, ordered, partially-ordered and reverse-ordered
 */
@Test
public void sortDiffStatusArrays() throws IOException {
    final Random random = new Random();
    //Run m times
    int m = 100;

    Integer[] arr = InsertionSort.initialArr(InsertionSort.RANDOM_ARRAY,
100).get();
    int len = 1000;  //n
    for (int i = 0; i < 6; i++) {
        double randomTime =
InsertionSort.runBenchmarkTimer(InsertionSort.initialArr(InsertionSort.RAND
OM_ARRAY, len), m);
        double orderedTime =
InsertionSort.runBenchmarkTimer(InsertionSort.initialArr(InsertionSort.ORDE
RED_ARRAY, len), m);
        double partiallyOrderedTime =
InsertionSort.runBenchmarkTimer(InsertionSort.initialArr(InsertionSort.PART
IALLY_ORDERED_ARRAY, len), m);
        double reverseOrderedTime =
InsertionSort.runBenchmarkTimer(InsertionSort.initialArr(InsertionSort.REVE
RSE_ORDERED_ARRAY, len), m);
```

```
            System.out.println("Arr length:     Random:      \t     Ordered:
        Partially Ordered:    Reversed Ordered:");
            System.out.printf("\t %d \t %.12f \t %.12f \t %.12f \t %.12f", len,
    randomTime, orderedTime, partiallyOrderedTime, reverseOrderedTime);
            System.out.println();
            len *= 2;
        }
    }
```

4. Evidence

4.1 Timer

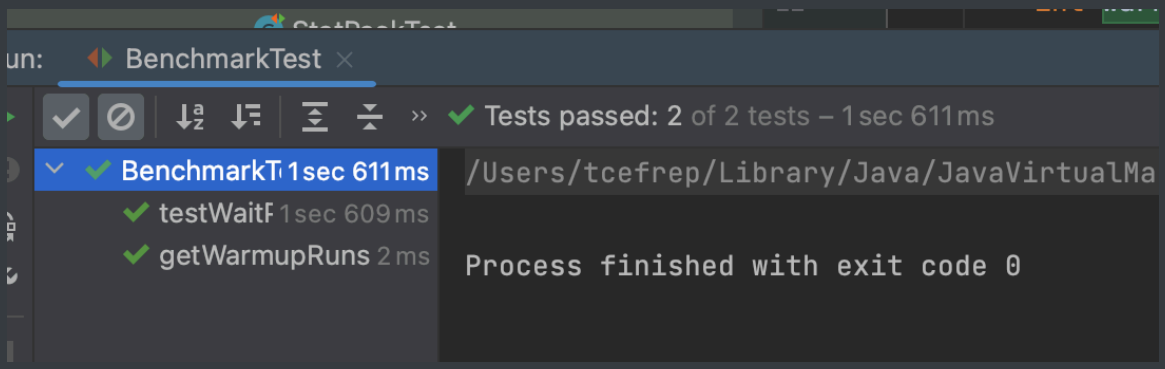

Figure 1: Sceenshots of Benchmark Test

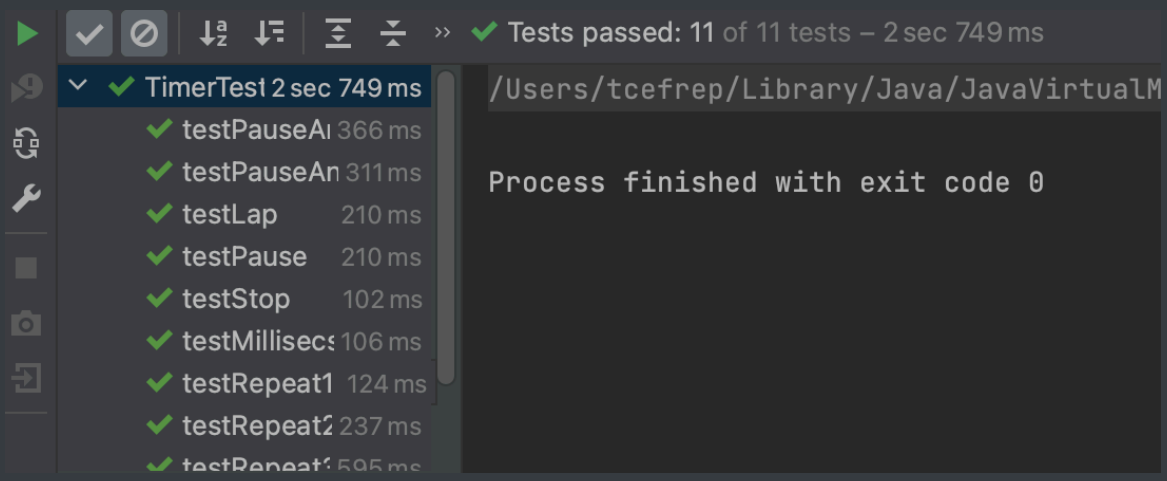

Figure 2: Sceenshots of Timer Test

4.2 InsertionTest

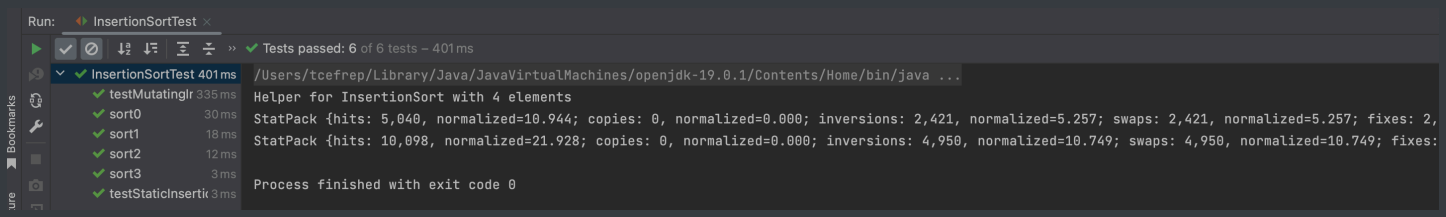

Figure 3: Sceenshots of Insertion Test

4.3 Calculate the time of four various arrays

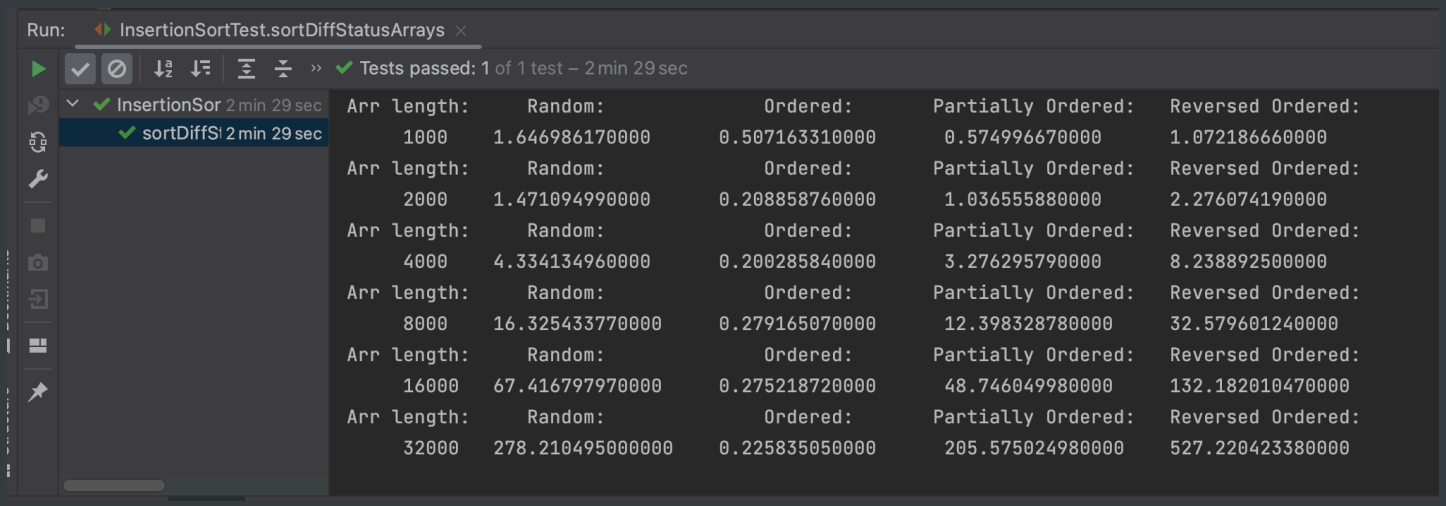

Figure 4: Sceenshots of timing four different arrys by using insertion sort

5. Conclusion

```java
10 usages  xiaohuanlin *
@Override
public double runFromSupplier(Supplier<T> supplier, int m) {
//        logger.info("Begin run: " + description + " with " + formatWhole(m) + " runs");
    // Warmup phase
    final Function<T, T> function = t -> {
        fRun.accept(t);
        return t;
    };
    new Timer().repeat(getWarmupRuns(m), supplier, function, fPre,  postFunction: null);

    // Timed phase
    return new Timer().repeat(m, supplier, function, fPre, fPost);
}
```

In order to ensure more accurate experimental results, this method also has "warmed up" function before really executing business logic.

From the screenshots of chapter 4.3. We can draw the following relationship:

> Time spent sorting the array by using insertion sort:
>
> **Reverse Ordered > Random > Partially Ordered > Ordered**
>
> - Since the time complexity of insertion sort is **O(n^2)**, we notice that every time when we double the array's length, we take about four times (approximately equal to n^2) as long as before in sorting **Random array, Partially Ordered array, Reverse Ordered array**.
> - However, when we sort **ordered array** by adopting insertion sort, we just compare all of elements of the **ordered array**. Therefore, the sorting time does not change significantly with the increase of array length.

In fact, if we think about it carefully, it is not hard for us to think out this relationship. Each time we run the insertion sort, this method will put a smaller element before the larger one.

> For example: **In Reverse Ordered:**

> Array: 5, 4, 3, 2, 1
>
> First time: 5, 4, 3, 2, 1 -> 1, 5, 4, 3, 2
>
> Second time: 1, 5, 4, 3, 2 -> 1, 2, 5, 4, 3
>
> .....
>
> Since it is sorted in reverse order, it puts the last element (smallest one in unordered element) to the first place of the unordered element and move other elements back. However, in **Ordered** array, insertion sort would only compare element instead of swapping elements, since all elements are in the correct position. Therefore, **Ordered** array spend least time in insertion sort.