Akash Bharadwaj Karthik (002787011)

Program Structures & Algorithms
Spring 2023 (Sec -8)
Assignment No. 3

Task

- (Part 1) You are to implement three (3) methods (*repeat*, *getClock*, and *toMillisecs*) of a class called *Timer*. Also run the Benchmark_Timer and Timer unit tests.

- (Part 2) Implement *InsertionSort* (in the *InsertionSort* class) by simply looking up the insertion code used by *Arrays.sort. Also runs its unit tests.*

- (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.

Relationship Conclusion:

The order of growth of the running time of Insertion Sort (Randomly ordered array of size *N*) is found to be $\approx N^{1.92}$.

The order of growth of the running time of Insertion Sort (Ordered array of size *N*) is found to be $\approx N^{1.08}$.

The order of growth of the running time of Insertion Sort (Partially ordered array of size *N*) is $\approx N^{1.903}$.

The order of growth of the running time of Insertion Sort (Reverse ordered array of size *N*) is $\approx N^{2.10}$.

Insertion Sort Time Complexity in order of increasing growth rates can be classified based on the condition of the array as follows:

$$Ordered < Partially\ Ordered < Randomly\ Ordered < Reverse\ Ordered$$

Evidence to support the conclusion:

A table of values recording array size - 'N', number of runs - 'runs', and time in milliseconds - 'time', is observed for insertion sort on each type of array, and slope from log-log plot is generated to approximate the time complexity for each input scenario:

We already know from theoretical analysis of insertion sort that only polynomial terms are involved in its time complexity, and hence that $T(n)$ is of the form:

$$T(N) = aN^b$$

Where,

N = input size

a = machine dependent constant

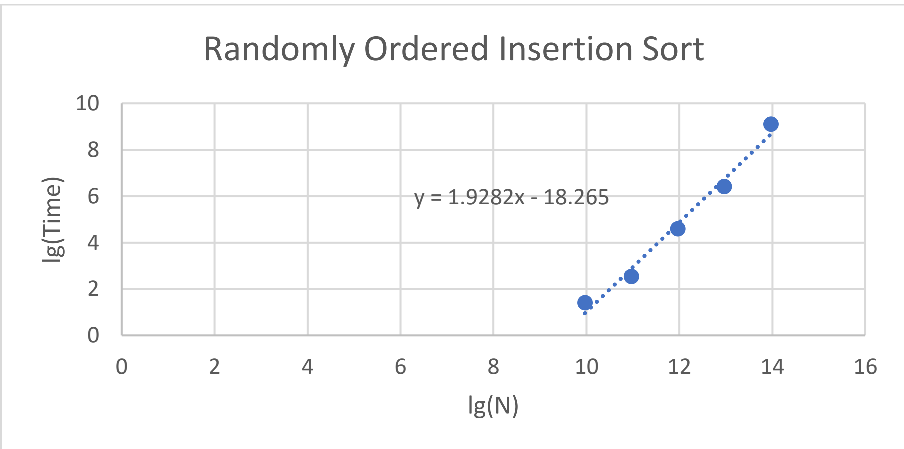
b = slope of the log-log scale graph

Therefore, the slope of the lg-lg plot will be used to approximate the power 'b'.

Values of 'N' are doubled, and increment as follows: 1000,2000,4000,8000,16000.

N	Runs	Randomly Ordered			
		Time(milliseconds)	lg(Time)	lg(N)	Slope
1000	30	2.64	1.40053793	9.965784285	-
2000	30	5.78	2.531069493	10.96578428	1.130531563
4000	30	24.06	4.588564737	11.96578428	2.057495245
8000	30	85.39	6.415995221	12.96578428	1.827430484
16000	30	548.4	9.099084761	13.96578428	2.68308954

Analysis of experimental data (the running time of insertion sort with random ordered input)

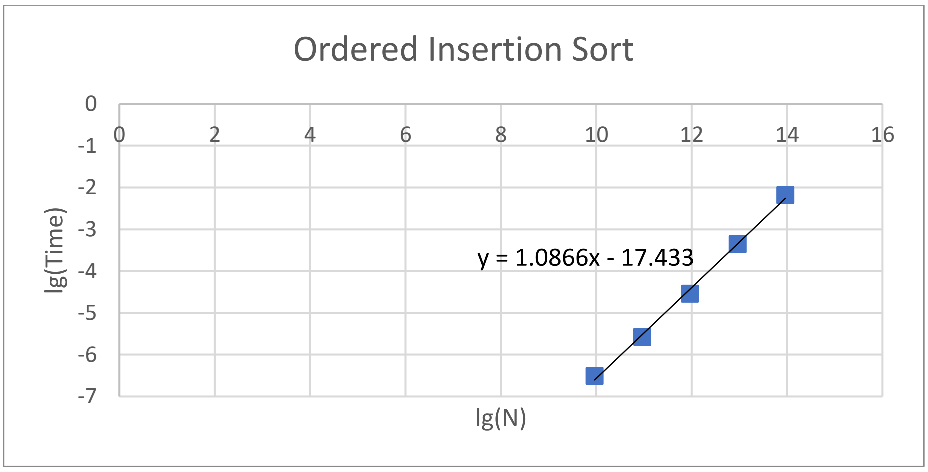


lg(N)-lg(Time) plot for insertion sort on randomly ordered array

Hence, for randomly sorted array : T(n) ∝ $N^{1.92}$ since the slope is approximately 1.92

N	Runs	Ordered			
		Time(milliseconds)	lg(Time)	lg(N)	Slope
1000	30	0.011	-6.506352666	9.965784285	-
2000	30	0.021	-5.573466862	10.96578428	0.932885804
4000	30	0.043	-4.53951953	11.96578428	1.033947332
8000	30	0.098	-3.351074441	12.96578428	1.188445089
16000	30	0.22	-2.184424571	13.96578428	1.166649869

Analysis of experimental data (the running time of insertion sort with an ordered input)

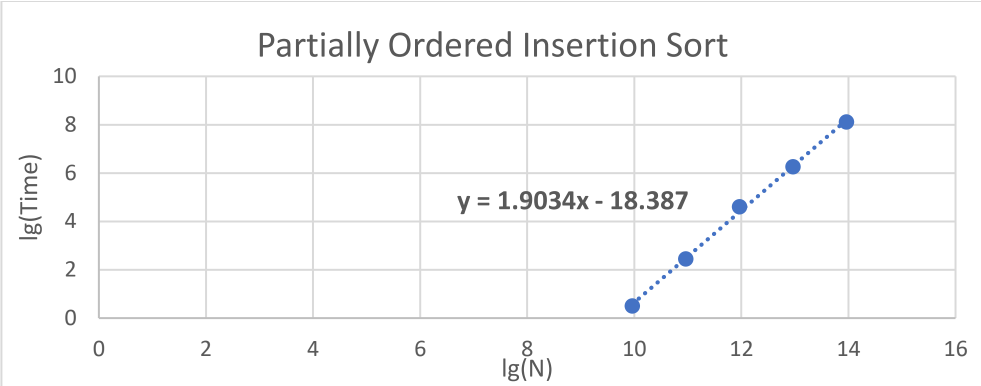


lg(N)-lg(Time) plot for insertion sort on ordered array

Hence, for array in order : T(n) ∝ $N^{1.08}$ since the slope is approximately 1.08

N	Runs	Partially Ordered			
		Time(milliseconds)	lg(Time)	lg(N)	Slope
1000	30	1.42	0.50589093	9.965784285	-
2000	30	5.47	2.451540833	10.96578428	1.945649903
4000	30	24.32	4.604071324	11.96578428	2.152530491
8000	30	76.95	6.265849421	12.96578428	1.661778098
16000	30	277.38	8.115719958	13.96578428	1.849870537

Analysis of experimental data (the running time of insertion sort with partially ordered input)

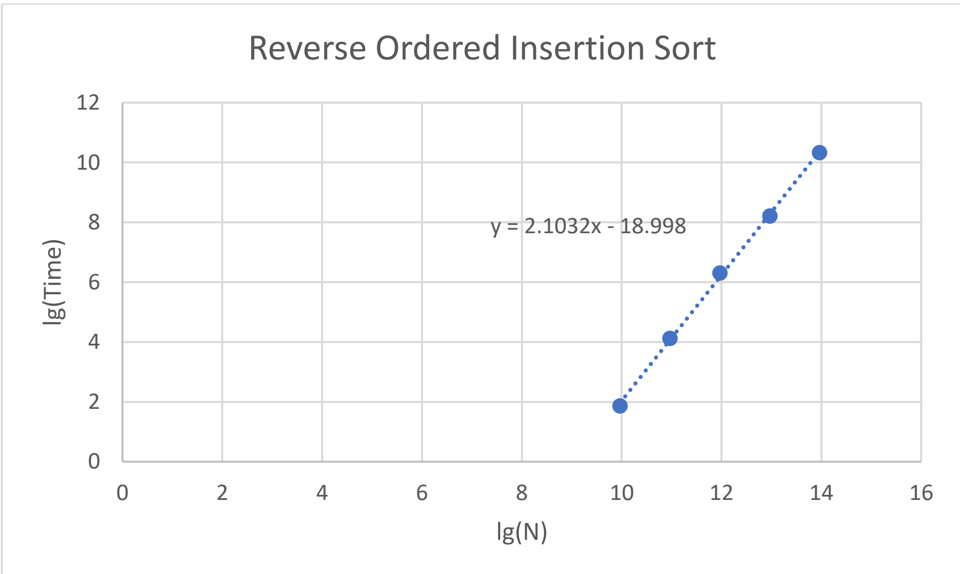


lg(N)-lg(Time) plot for insertion sort on partially ordered array

Hence, for partially sorted array : T(n) $\propto$ $N^{1.90}$ since the slope is approximately 1.90

N	Runs	Reverse Ordered			
		Time(milliseconds)	lg(Time)	lg(N)	Slope
1000	30	3.64	1.86393845	9.965784285	-
2000	30	17.43	4.123500664	10.96578428	2.259562214
4000	30	79.09	6.305423389	11.96578428	2.181922725
8000	30	296.8	8.213347282	12.96578428	1.907923892
16000	30	1291.54	10.33487661	13.96578428	2.121529328

Analysis of experimental data (the running time of insertion sort with reverse ordered input)



lg(N)-lg(Time) plot for insertion sort on reverse ordered array

Hence, for reverse ordered array : T(n) $\propto$ $N^{2.10}$ since the slope is approximately 2.10

Analysis:

Insertion Sort with random sorted data:

We expect that there will be an average of $\frac{1}{2}\binom{n}{2}$ inversions when elements are randomly ordered, and hence, insertion sort will need to make at least $\frac{n^2}{4}$ compares, and so we expect a quadratic time complexity. The time complexity term observed scaling with $\boldsymbol{N^{1.92}}$ is a reasonable figure close to quadratic.

Insertion Sort with ordered data:

We expect that there will be N – 1 compares and 0 swaps when the data is completely ordered, hence a linear time complexity is expected; the time complexity term observed scaling with $\boldsymbol{N^{1.08}}$ is a reasonable figure close to linear.

Insertion Sort with partially ordered data:

We expect that there will be half the number of inversion of the average case of randomly ordered data, and hence a quadratic expression for time complexity is still expected. The time complexity term observed scaling with $\boldsymbol{N^{1.90}}$ is a reasonable figure close to quadratic.

Insertion Sort with reverse ordered data:

We expect that there will maximum number of comparisons $(\frac{n^2}{4} + n - 1)$ and swaps, and hence this is likely to be the highest growing time complexity term and also asymptotically quadratic. The time complexity term observed scaling with $\boldsymbol{N^{2.10}}$ is a reasonable figure close to quadratic.

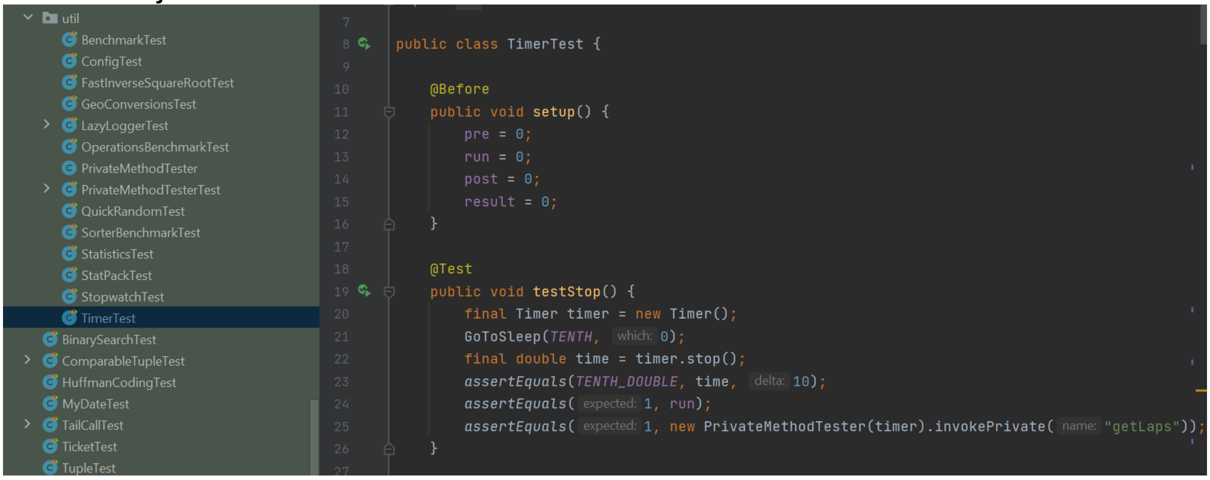
Console Output:

```
RANDOMLY ORDERED ARRAY
2023-02-04 22:12:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 1000, Time Taken: 2.6423566666666667
2023-02-04 22:12:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 2000, Time Taken: 5.78663
2023-02-04 22:12:50 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 4000, Time Taken: 24.068253333333335
2023-02-04 22:12:50 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 8000, Time Taken: 85.39465666666666
2023-02-04 22:12:53 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 16000, Time Taken: 548.4040666666666

ORDERED ARRAY
2023-02-04 22:13:11 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 1000, Time Taken: 0.011013333333333335
2023-02-04 22:13:11 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 2000, Time Taken: 0.021086666666666667
2023-02-04 22:13:11 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 4000, Time Taken: 0.04308
2023-02-04 22:13:11 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 8000, Time Taken: 0.09824999999999999
2023-02-04 22:13:11 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 16000, Time Taken: 0.21955333333333332
```

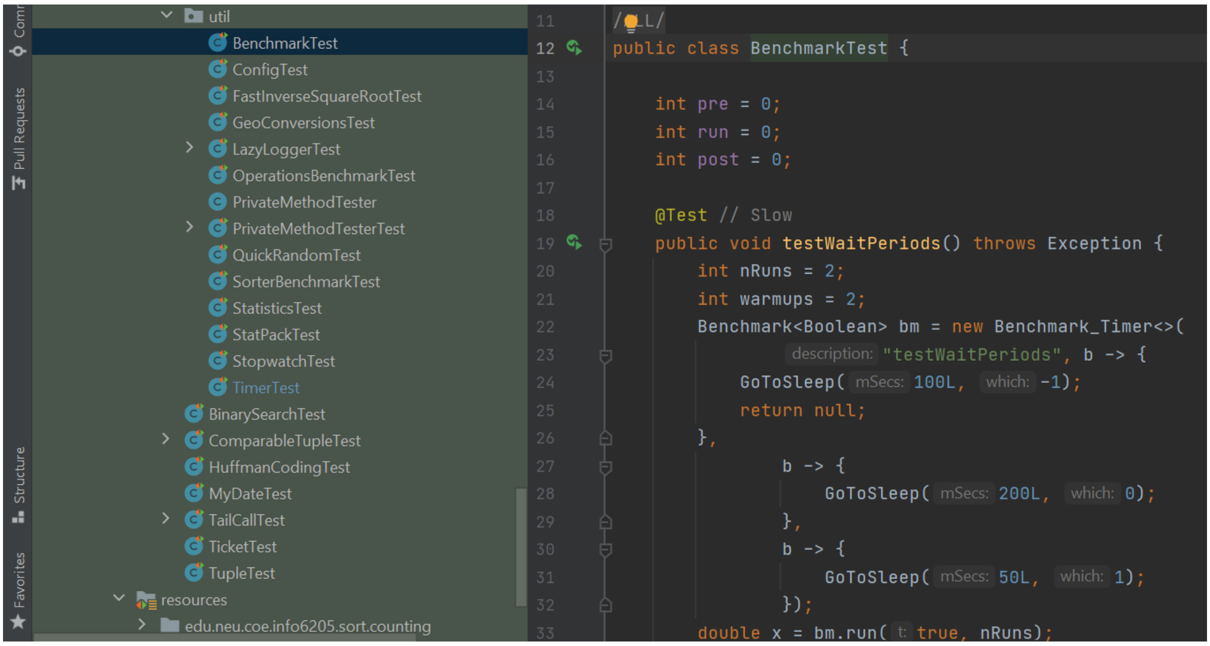
```
PARTIALLY ORDERED ARRAY
2023-02-04 22:13:12 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 1000, Time Taken: 1.4183733333333335
2023-02-04 22:13:12 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 2000, Time Taken: 5.470353333333334
2023-02-04 22:13:12 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 4000, Time Taken: 24.322606666666665
2023-02-04 22:13:13 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 8000, Time Taken: 76.94668666666666
2023-02-04 22:13:16 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 16000, Time Taken: 277.3846566666667

REVERSE ORDERED ARRAY
2023-02-04 22:13:25 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 1000, Time Taken: 3.6351133333333334
2023-02-04 22:13:25 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 2000, Time Taken: 17.426243333333332
2023-02-04 22:13:26 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 4000, Time Taken: 79.08924666666667
2023-02-04 22:13:28 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 8000, Time Taken: 296.7940433333333
2023-02-04 22:13:38 INFO  Benchmark_Timer - Begin run: Insertion Sort with 30 runs
N= 16000, Time Taken: 1291.5417466666665
```

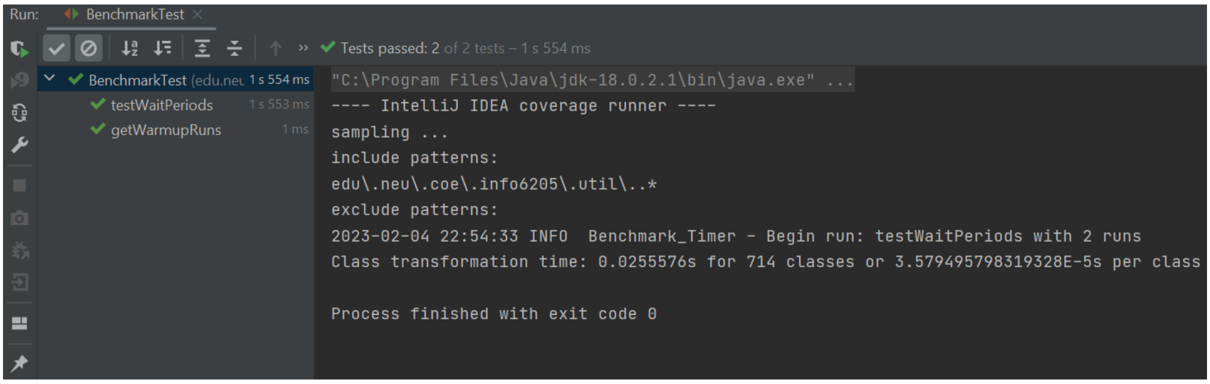
Unit Tests:

TimerTest.java

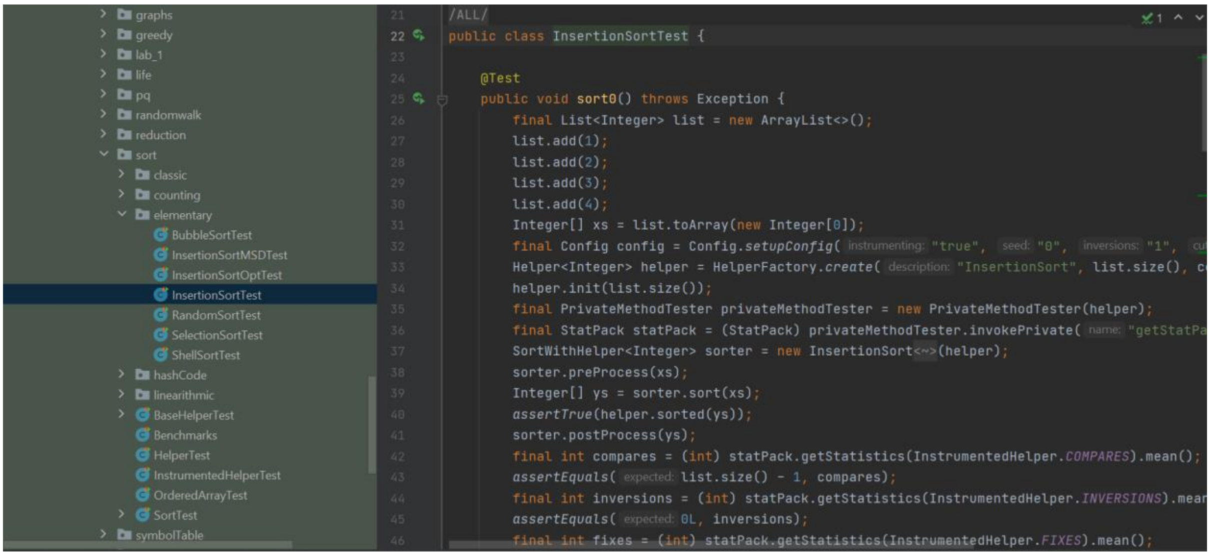


BenchmarkTest.java

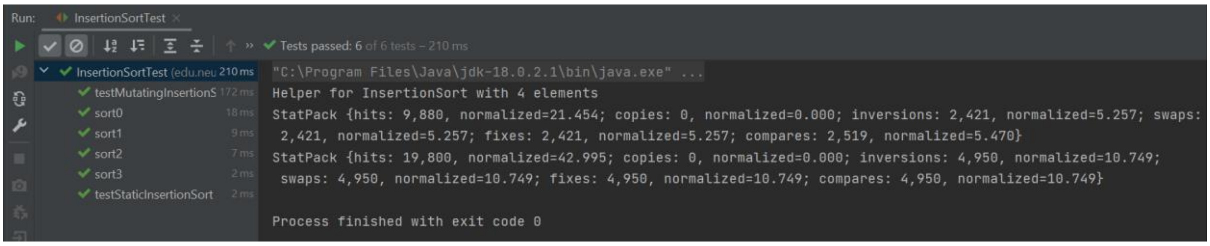




InsertionSortTest.java




```java
/ALL/
public class InsertionSortTest {

    @Test
    public void sort0() throws Exception {
        final List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        Integer[] xs = list.toArray(new Integer[0]);
        final Config config = Config.setupConfig( instrumenting: "true",  seed: "0",  inversions: "1",  cu
        Helper<Integer> helper = HelperFactory.create( description: "InsertionSort", list.size(), c
        helper.init(list.size());
        final PrivateMethodTester privateMethodTester = new PrivateMethodTester(helper);
        final StatPack statPack = (StatPack) privateMethodTester.invokePrivate( name: "getStatPa
        SortWithHelper<Integer> sorter = new InsertionSort<~>(helper);
        sorter.preProcess(xs);
        Integer[] ys = sorter.sort(xs);
        assertTrue(helper.sorted(ys));
        sorter.postProcess(ys);
        final int compares = (int) statPack.getStatistics(InstrumentedHelper.COMPARES).mean();
        assertEquals( expected: list.size() - 1, compares);
        final int inversions = (int) statPack.getStatistics(InstrumentedHelper.INVERSIONS).mear
        assertEquals( expected: 0L, inversions);
        final int fixes = (int) statPack.getStatistics(InstrumentedHelper.FIXES).mean();
```




```
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe" ...
Helper for InsertionSort with 4 elements
StatPack {hits: 9,880, normalized=21.454; copies: 0, normalized=0.000; inversions: 2,421, normalized=5.257; swaps:
 2,421, normalized=5.257; fixes: 2,421, normalized=5.257; compares: 2,519, normalized=5.470}
StatPack {hits: 19,800, normalized=42.995; copies: 0, normalized=0.000; inversions: 4,950, normalized=10.749;
 swaps: 4,950, normalized=10.749; fixes: 4,950, normalized=10.749; compares: 4,950, normalized=10.749}

Process finished with exit code 0
```