Program Structures and Algorithms
Spring 2024

NAME: Suchita Arvind Dabir
NUID: 002957879
GITHUB LINK: https://github.com/suchitadabir/INFO6205

Task

(Part 1) To implement three methods in the Timer class repeat, getClock, and toMillisecs.
(Part 2) Implement the InsertionSort class and ensure correctness through unit tests.
(Part 3) Execute the benchmarks to measure the running time of sort across four different initial array orderings: random, ordered, partially ordered, and reverse ordered.

Conclusion

On average, the time taken by different ordering methods are,

Scenario	Time Complexity	Description
Random Order	O(n^2)	Extensive comparisons and swaps required, resulting in quadratic time complexity.
Ordered (Best Case)	O(n)	Elements are already sorted, leading to linear time complexity.
Partially Ordered (Varies with degree of ordering)	Approaches O(n)	Nearly sorted arrays approach O(n).
	Approaches O(n^2)	Less ordered arrays approach O(n^2)
Reverse-Ordered (Worst Case)	O(n^2)	Extensive comparisons and swaps required, resulting in quadratic time complexity.

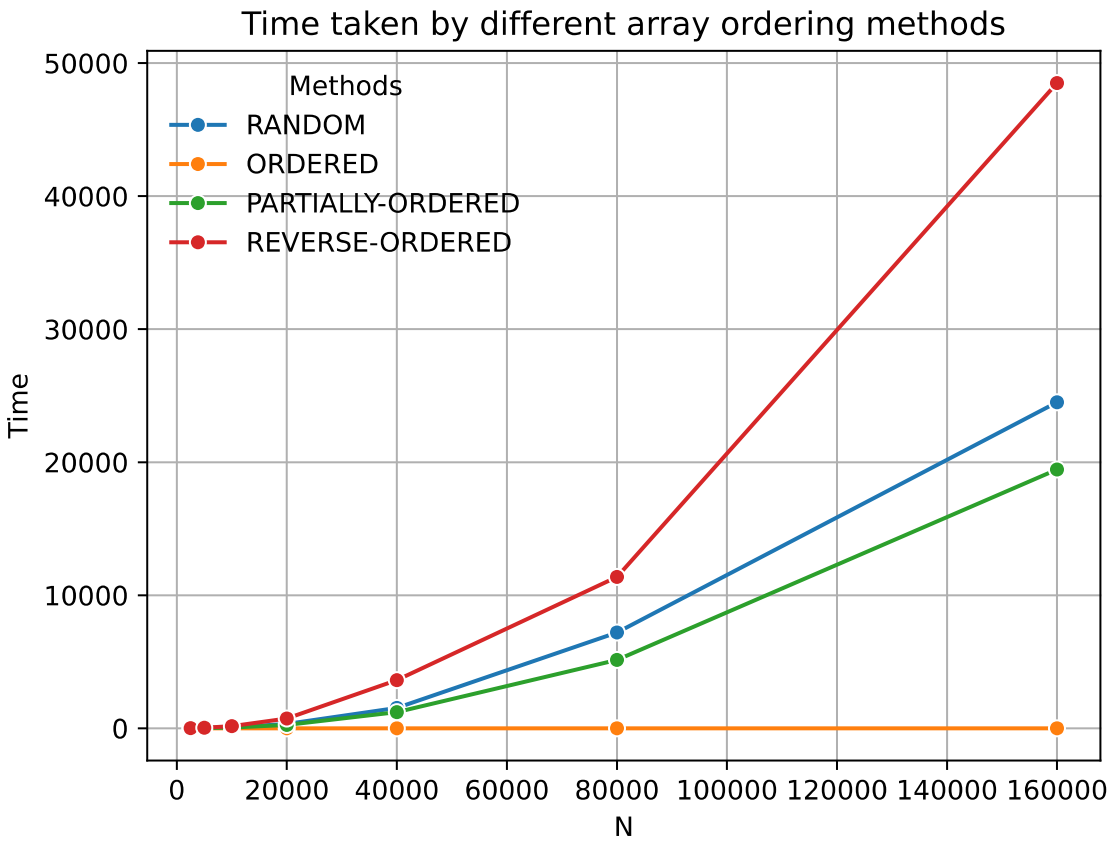
Evidence to support that conclusion:

- I utilized the Benchmark_Timer and Timer classes from the repository to measure the runtime of the InsertionSort algorithm across four different initial array ordering scenarios.
- In the Assignment3Benchmark class, I conducted benchmarks by choosing seven values of N using the doubling method and a safetyFactor set to 10 to ensure enough elements across various array ordering scenarios.
- The final values of N are 2500, 5000, 10000, 20000, 40000, 80000, and 160000.
- Subsequently I exported all data to a CSV file viz. Assignment3Benchmark.csv.
- The table below displays the runtime in milliseconds for each ordering scenarios at the selected value of N for various Run counts.
- For each invocation of run, given target function has run 10 times to get the system "warmed up" before starting the timing properly.

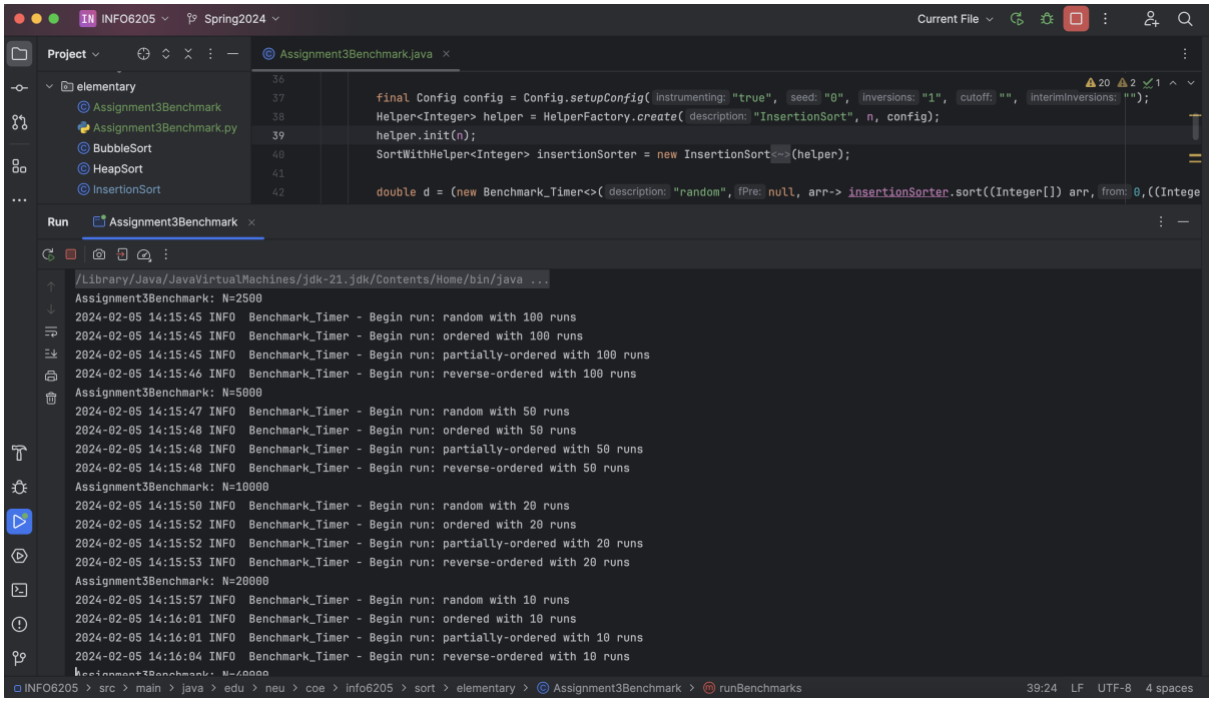
Data for run times of Insertion sort :

Array-Ordering	Runs	N	Time
RANDOM	100	2500	4.81896256
ORDERED	100	2500	0.00637838
PARTIALLY-ORDERED	100	2500	3.57540998
REVERSE-ORDERED	100	2500	9.40005659
RANDOM	50	5000	18.6252001
ORDERED	50	5000	0.01175076
PARTIALLY-ORDERED	50	5000	14.07222332
REVERSE-ORDERED	50	5000	37.980928400000000
RANDOM	20	10000	75.6887229
ORDERED	20	10000	0.02348335
PARTIALLY-ORDERED	20	10000	57.674806150000000
REVERSE-ORDERED	20	10000	159.17852935000000
RANDOM	10	20000	324.2387917
ORDERED	10	20000	0.0476916
PARTIALLY-ORDERED	10	20000	250.87678340000000
REVERSE-ORDERED	10	20000	735.5017625
RANDOM	5	40000	1533.9784
ORDERED	5	40000	0.10221660000000000
PARTIALLY-ORDERED	5	40000	1208.7346918000000
REVERSE-ORDERED	5	40000	3611.9991336
RANDOM	3	80000	7202.299722
ORDERED	3	80000	0.23293133333333300
PARTIALLY-ORDERED	3	80000	5138.769402333330
REVERSE-ORDERED	3	80000	11378.536306
RANDOM	2	160000	24507.034416
ORDERED	2	160000	0.70675
PARTIALLY-ORDERED	2	160000	19449.845125
REVERSE-ORDERED	2	160000	48501.4422295

- I wrote a simple python script to plot the run time in millisecond along Y-axis and N across X-axis.
- Below plotted graph shows the performance of the Insertion Sort algorithm on arrays with different initial conditions:
  - Random: Time increases at a likely quadratic rate as we increase the size of an array.
  - Ordered: It involves O(n) comparisons and O(1) swaps leading to O(n) overall time complexity resulting into the best-case scenario for Insertion sort.
  - Partially-Ordered: Time increases at a rate between the best and worst cases.
  - Reverse-Ordered: Time increases sharply, rate as we increase the size of an array. This shows a quadratic time complexity, which is the worst case for Insertion Sort.
- Hence, experiments suggest that Insertion Sort is fastest on already sorted data and slowest on reverse-sorted data. Its performance degrades significantly as the array size grows.

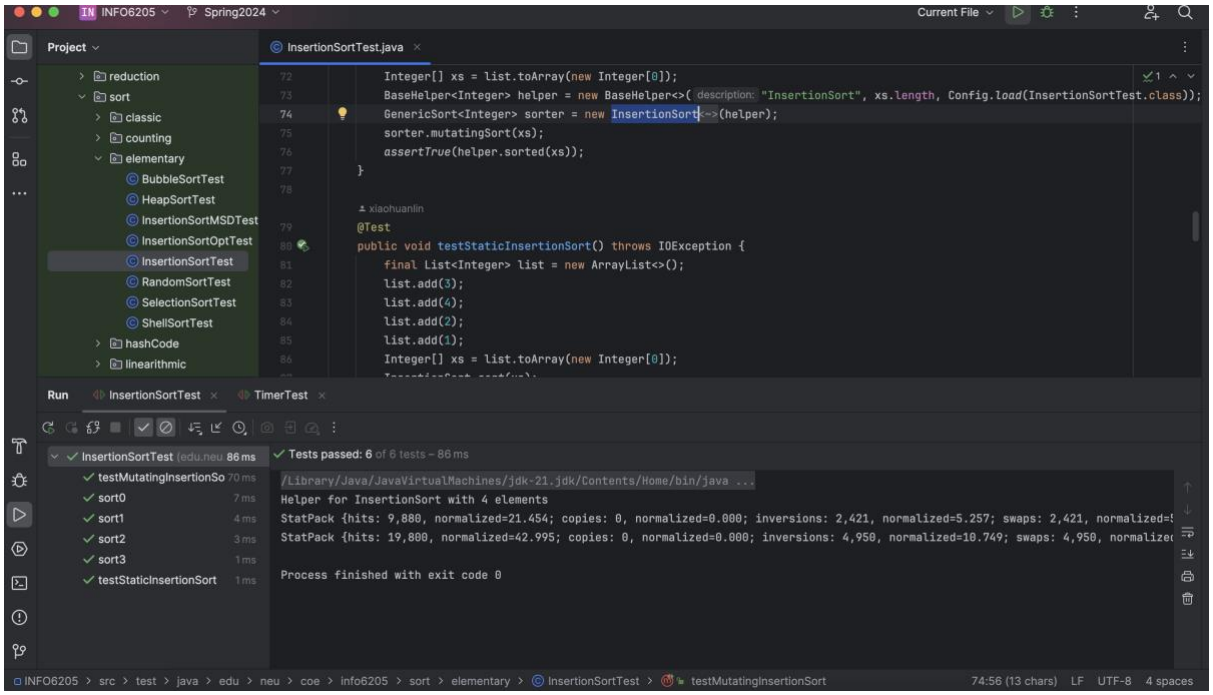


Runs Screenshot



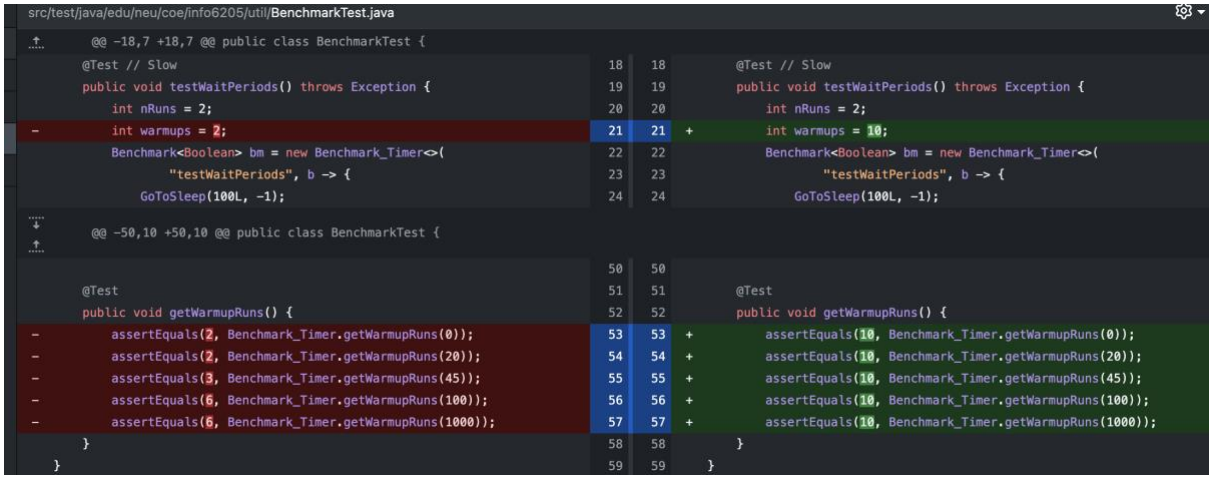
Unit Test Screenshots

InsertionSortTest – 6 of 6 tests passed.

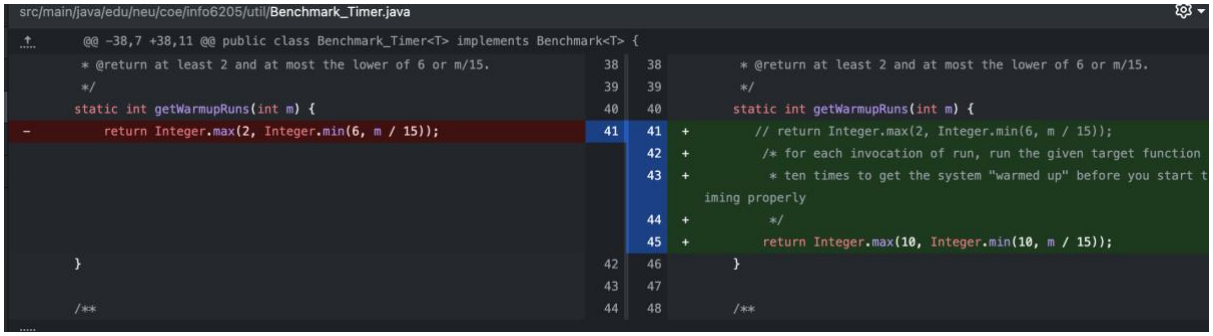


Changes made in BenchmarkTest.java and Benchmark_Timer.java to achieve following requirement:
For each invocation of run, run the given target function 10 times to get the system "warmed up" before starting the timing properly.

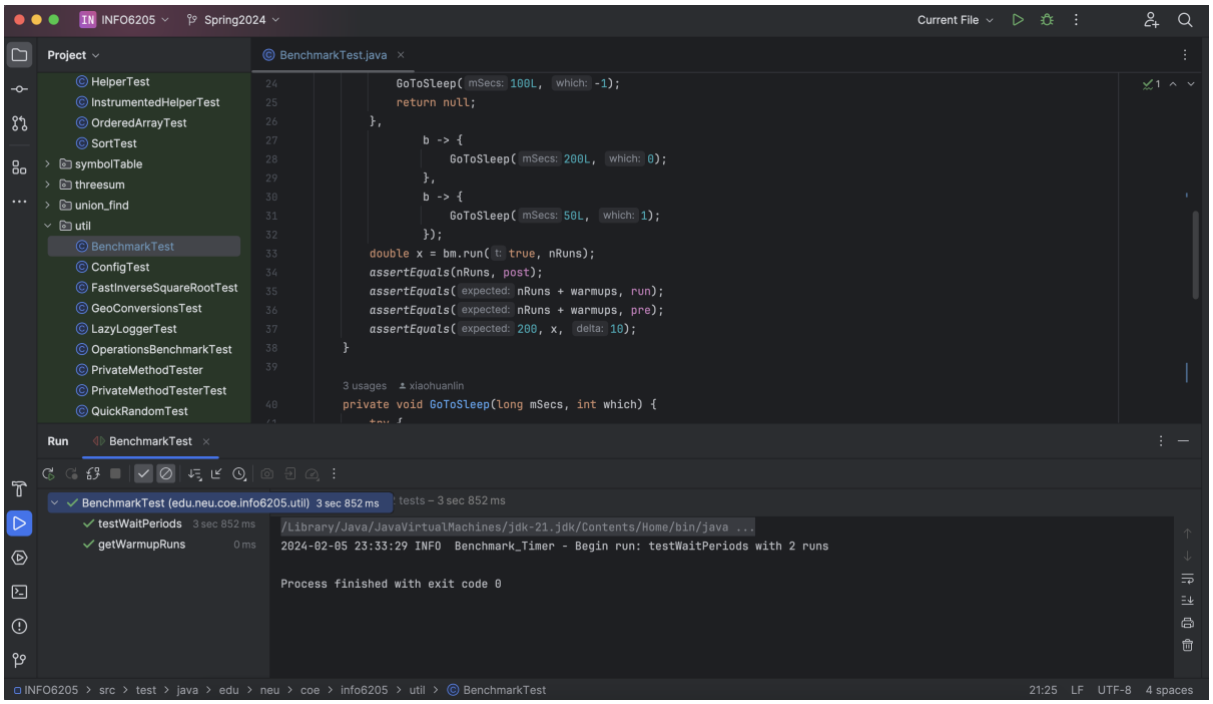
BenchmarkTest.java



BenchmarkTimer.java



BenchmarkTest [With warmup count as 10] – 2 of 2 Tests passed.



TimerTest – 11 of 11 tests passed.