Program Structures and Algorithms
Spring 2024

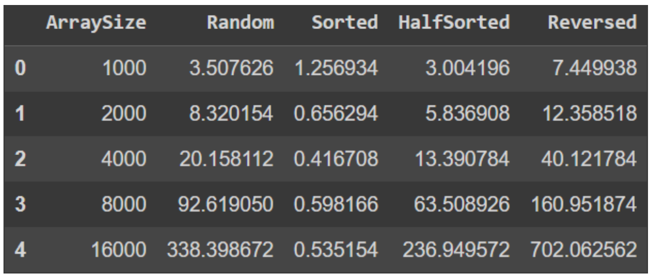
NAME: Mihir Ravindra Adelkar
NUID: 002810839
GITHUB LINK: https://github.com/mihiradelkar/INFO6205

Task:

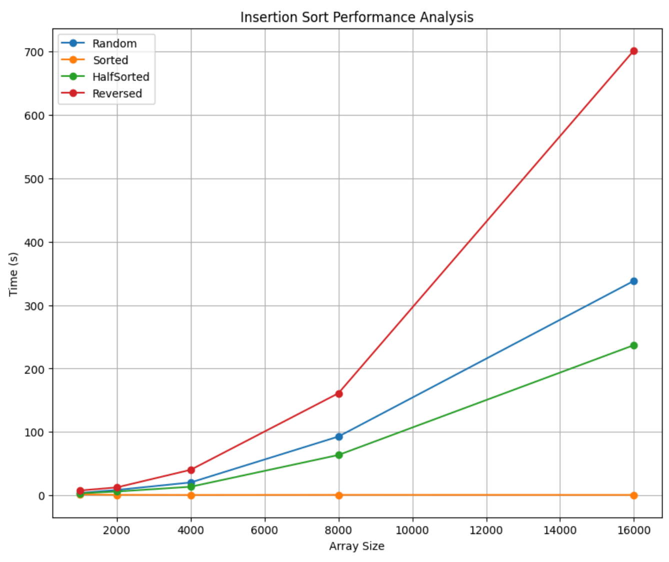
1. Implement three (3) methods (repeat, getClock, and toMillisecs) of a class called Timer. Please see the skeleton class that I created in the repository. Timer is invoked from a class called Benchmark_Timer which implements the Benchmark interface.
2. Implement InsertionSort by simply looking up the insertion code used by Arrays.sort. 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.
3. Implement a main program 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.

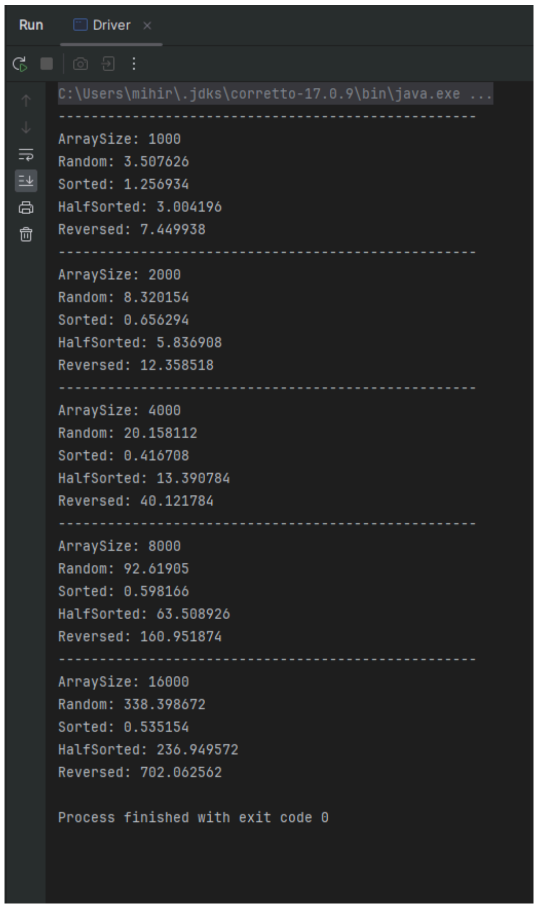
Relationship Conclusion:

The benchmark results and the derived equations for Insertion Sort under various array conditions (Random, Sorted, HalfSorted, Reversed) demonstrate the algorithm's performance characteristics and scalability. The relationship between array size and sorting time is significantly influenced by the initial order of elements in the array. Specifically, the performance of Insertion Sort is optimal for already sorted arrays, where its time complexity approaches O(n), and notably less efficient for reversed and random arrays, where the performance degrades towards O(n^2).



	ArraySize	Random	Sorted	HalfSorted	Reversed
0	1000	3.507626	1.256934	3.004196	7.449938
1	2000	8.320154	0.656294	5.836908	12.358518
2	4000	20.158112	0.416708	13.390784	40.121784
3	8000	92.619050	0.598166	63.508926	160.951874
4	16000	338.398672	0.535154	236.949572	702.062562





```
C:\Users\mihir\.jdks\corretto-17.0.9\bin\java.exe ...
-----------------------------------------------
ArraySize: 1000
Random: 3.507626
Sorted: 1.256934
HalfSorted: 3.004196
Reversed: 7.449938
-----------------------------------------------
ArraySize: 2000
Random: 8.320154
Sorted: 0.656294
HalfSorted: 5.836908
Reversed: 12.358518
-----------------------------------------------
ArraySize: 4000
Random: 20.158112
Sorted: 0.416708
HalfSorted: 13.390784
Reversed: 40.121784
-----------------------------------------------
ArraySize: 8000
Random: 92.61905
Sorted: 0.598166
HalfSorted: 63.508926
Reversed: 160.951874
-----------------------------------------------
ArraySize: 16000
Random: 338.398672
Sorted: 0.535154
HalfSorted: 236.949572
Reversed: 702.062562

Process finished with exit code 0
```

Evidence to support that conclusion:

Random Arrays:
As the array size doubles, the time taken for sorting increases more than linearly, which is expected due to the $O(n^2)$ time complexity of Insertion Sort for random arrays. For instance, the time increases from 3.51s for an array of size 1000 to 338.4s for an array of size 16000.

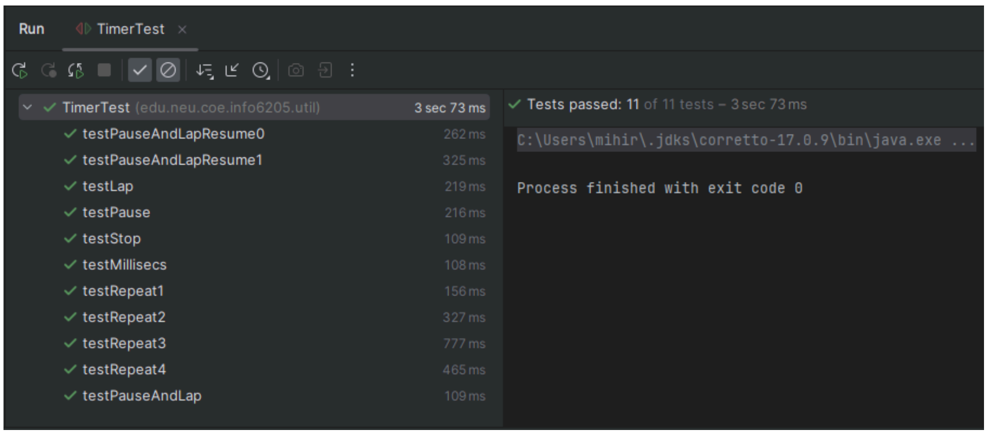
Sorted Arrays:
The time taken is significantly less for arrays that are already sorted. This is the best-case scenario for Insertion Sort, with a complexity close to $O(n)$ because it only needs to make a single comparison per element. Interestingly, the time slightly fluctuates but remains very low compared to other conditions, showing the efficiency of Insertion Sort for nearly or completely sorted data.

HalfSorted Arrays:
These arrays show an interesting pattern where the time taken is less than that for random arrays but significantly more than for sorted arrays. The increase in time as the array size doubles is substantial, indicating that the partially sorted portion does provide some efficiency gains, but the algorithm still leans towards quadratic time complexity for the unsorted portion.

Reversed Arrays:
This condition shows the worst performance, as expected, because every element needs to be moved to its correct position, showcasing the $O(n^2)$ complexity in the clearest form. The time taken more than doubles with each doubling of the array size, indicating the heavy penalty of having data in the worst possible order for Insertion Sort.

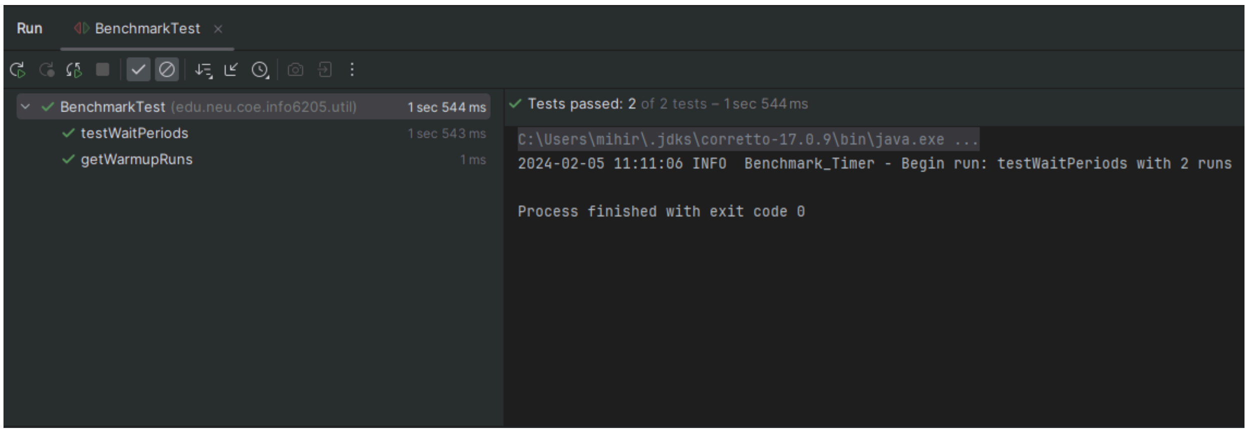
The pattern of these results firmly establishes the relationship between the initial order of the array and the efficiency of Insertion Sort. While the algorithm excels with data that is already sorted, requiring minimal computational effort, its performance degrades significantly with random or reverse-ordered data, where the inherent $O(n^2)$ complexity becomes pronounced. This emphasizes the importance of considering the initial data condition when choosing Insertion Sort for practical applications, highlighting its suitability for datasets that are already near or completely sorted.

Unit Test Screenshots:



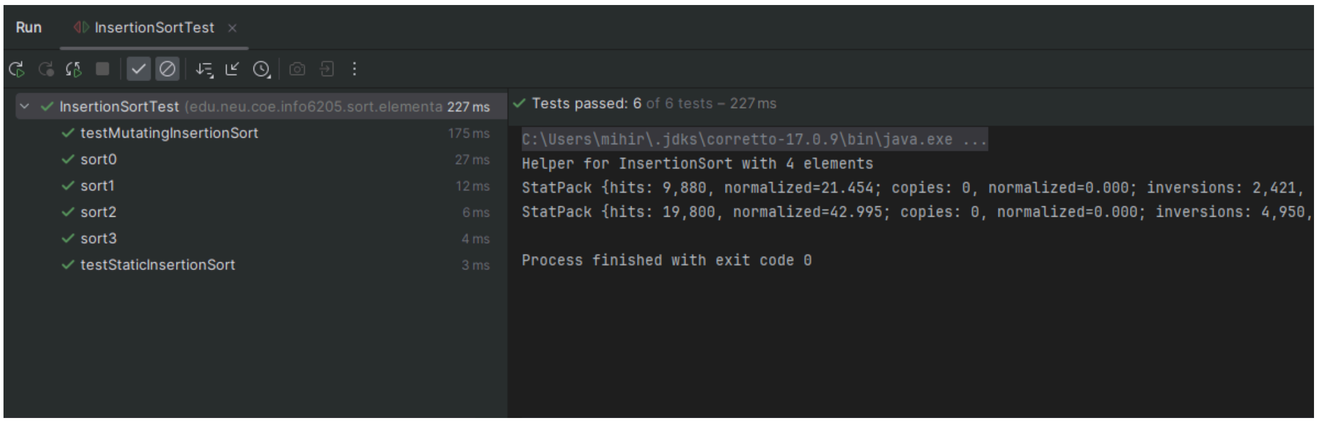
```
✓ TimerTest (edu.neu.coe.info6205.util)   3 sec 73 ms
    ✓ testPauseAndLapResume0              262 ms
    ✓ testPauseAndLapResume1              325 ms
    ✓ testLap                             219 ms
    ✓ testPause                           216 ms
    ✓ testStop                            109 ms
    ✓ testMillisecs                       108 ms
    ✓ testRepeat1                         156 ms
    ✓ testRepeat2                         327 ms
    ✓ testRepeat3                         777 ms
    ✓ testRepeat4                         465 ms
    ✓ testPauseAndLap                     109 ms
```

```
✓ Tests passed: 11 of 11 tests – 3 sec 73 ms

C:\Users\mihir\.jdks\corretto-17.0.9\bin\java.exe ...

Process finished with exit code 0
```



Run
BenchmarkTest
BenchmarkTest (edu.neu.coe.info6205.util) 1 sec 544 ms
testWaitPeriods 1 sec 543 ms
getWarmupRuns 1 ms
Tests passed: 2 of 2 tests – 1 sec 544 ms

```
C:\Users\mihir\.jdks\corretto-17.0.9\bin\java.exe ...
2024-02-05 11:11:06 INFO  Benchmark_Timer - Begin run: testWaitPeriods with 2 runs

Process finished with exit code 0
```



Run
InsertionSortTest
InsertionSortTest (edu.neu.coe.info6205.sort.elementa 227 ms
testMutatingInsertionSort 175 ms
sort0 27 ms
sort1 12 ms
sort2 6 ms
sort3 4 ms
testStaticInsertionSort 3 ms
Tests passed: 6 of 6 tests – 227 ms

```
C:\Users\mihir\.jdks\corretto-17.0.9\bin\java.exe ...
Helper for InsertionSort with 4 elements
StatPack {hits: 9,880, normalized=21.454; copies: 0, normalized=0.000; inversions: 2,421,
StatPack {hits: 19,800, normalized=42.995; copies: 0, normalized=0.000; inversions: 4,950,

Process finished with exit code 0
```