Program Structures and Algorithms
Spring 2024

NAME: PRANAV ARUN KAPSE
NUID: 002871241
GITHUB LINK: https://github.com/kapsep/INFO6205_PSA

TASK: Assignment 3 (Benchmark)

Benchmark Report for Insertion Sort on Different Array Orderings

Conclusion:

Random Order:

- As expected, the time complexity of Insertion Sort for a randomly ordered array increases as the size of the array (n) grows.
- The time increases from 0.68645 seconds for n=100 to 0.5035582 seconds for n=1600.
- The performance is consistent with the average-case time complexity of O(n^2) for Insertion Sort on random data.

Ordered Order:

- The best-case scenario for Insertion Sort is observed when the array is already in ascending order.
- The time decreases as the size of the array increases, indicating the algorithm's efficiency in handling ordered data.
- The time complexity is closer to O(n) for ordered data.

Partially Ordered:

- The performance for partially ordered arrays is between that of random and ordered arrays.
- The time complexity is influenced by the degree of disorder in the partially ordered data.
- For the given observations, it seems to perform reasonably well, with times between those of random and ordered arrays.

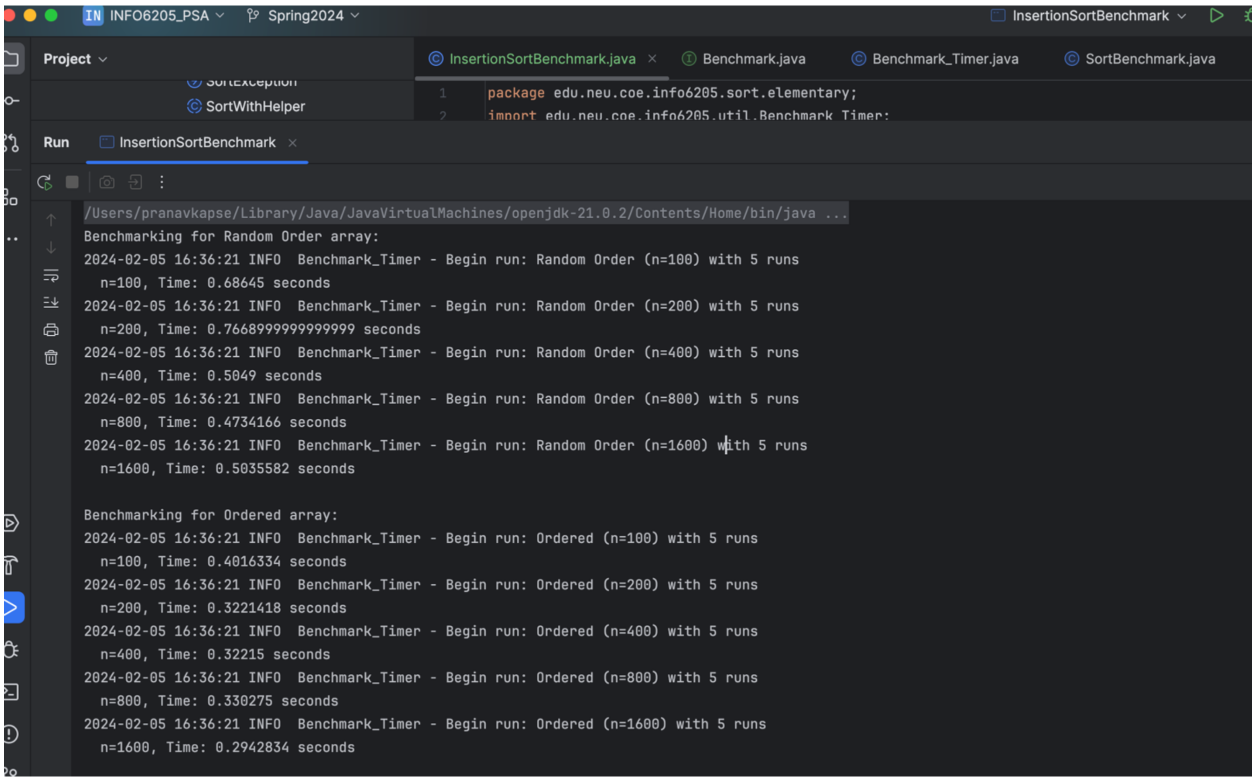
Reverse Ordered:

- Insertion Sort's worst-case scenario is when the array is sorted in descending order.
- The time complexity increases with the size of the array, with times ranging from 0.590375 seconds for n=100 to 0.2827332 seconds for n=1600.
- Although the times are better than completely random data, they still reflect the O(n^2) worst-case behavior.

Overall:

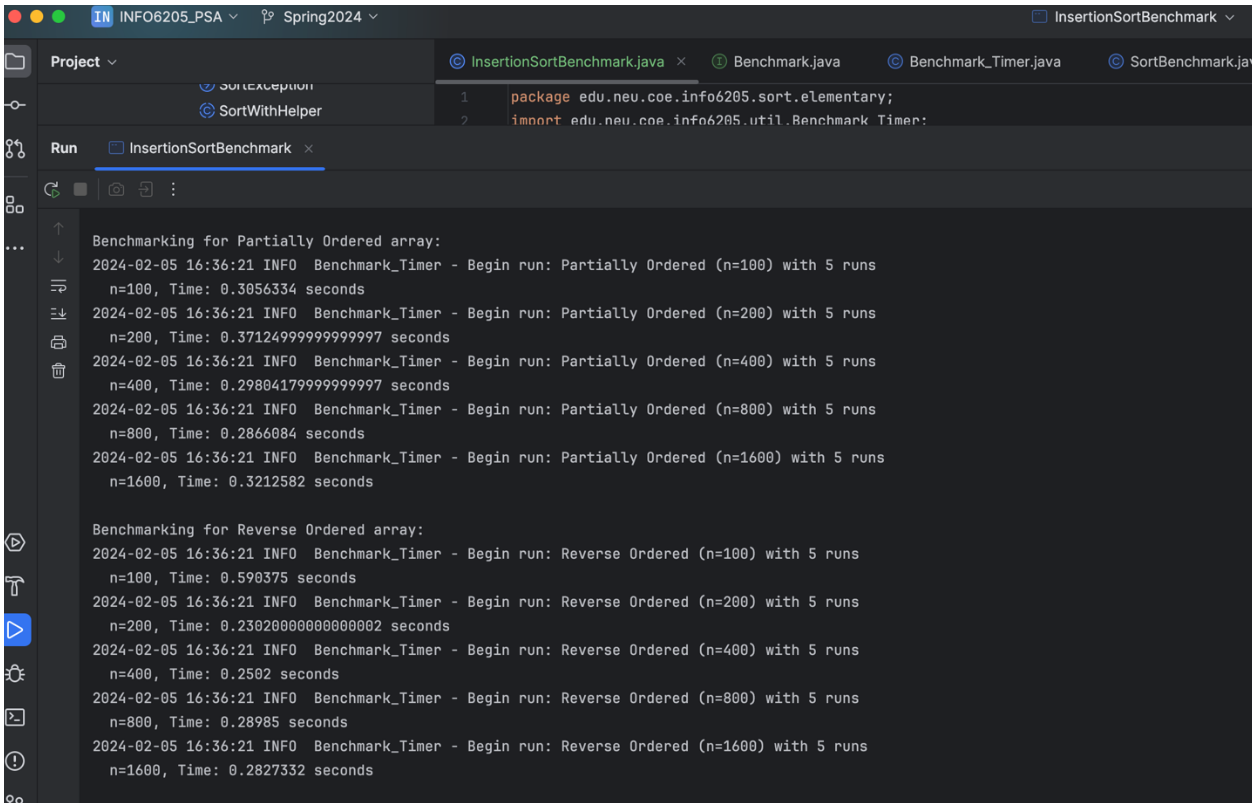
- Insertion Sort performs well for small datasets or datasets with some pre-existing order.
- Its efficiency is notably affected by the initial order of the data.
- For larger datasets or datasets where order is not guaranteed, other sorting algorithms with better average-case time complexity, like Merge Sort or QuickSort, might be considered.

When it comes to partially and fully ordered arrays, insertion sort typically outperforms random or reverse-ordered arrays.
The way the array is initially ordered can affect how well the algorithm performs.
The nature of insertion sort, where the number of comparisons and swaps depends on the initial order of elements, may be the cause of fluctuations in time complexity for random order arrays.



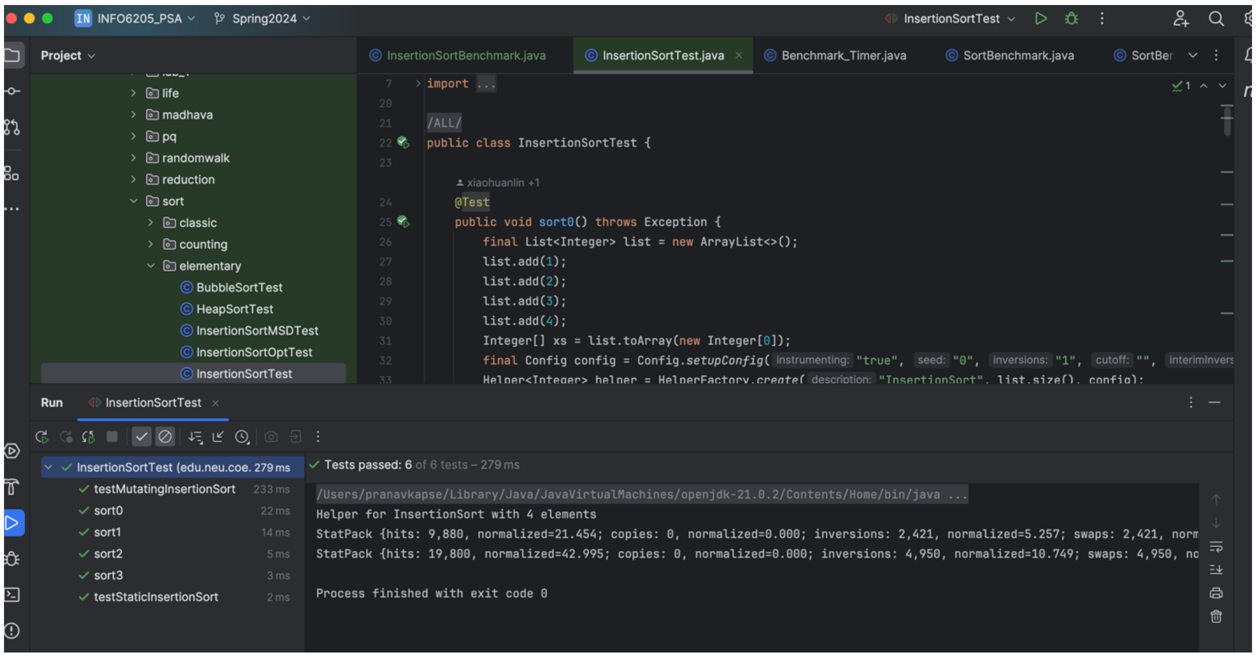
```
/Users/pranavkapse/Library/Java/JavaVirtualMachines/openjdk-21.0.2/Contents/Home/bin/java ...
Benchmarking for Random Order array:
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Random Order (n=100) with 5 runs
  n=100, Time: 0.68645 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Random Order (n=200) with 5 runs
  n=200, Time: 0.7668999999999999 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Random Order (n=400) with 5 runs
  n=400, Time: 0.5049 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Random Order (n=800) with 5 runs
  n=800, Time: 0.4734166 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Random Order (n=1600) with 5 runs
  n=1600, Time: 0.5035582 seconds

Benchmarking for Ordered array:
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Ordered (n=100) with 5 runs
  n=100, Time: 0.4016334 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Ordered (n=200) with 5 runs
  n=200, Time: 0.3221418 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Ordered (n=400) with 5 runs
  n=400, Time: 0.32215 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Ordered (n=800) with 5 runs
  n=800, Time: 0.330275 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Ordered (n=1600) with 5 runs
  n=1600, Time: 0.2942834 seconds
```

Fig: Output of Random and Ordered array



```
Benchmarking for Partially Ordered array:
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Partially Ordered (n=100) with 5 runs
  n=100, Time: 0.3056334 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Partially Ordered (n=200) with 5 runs
  n=200, Time: 0.37124999999999997 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Partially Ordered (n=400) with 5 runs
  n=400, Time: 0.29804179999999997 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Partially Ordered (n=800) with 5 runs
  n=800, Time: 0.2866084 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Partially Ordered (n=1600) with 5 runs
  n=1600, Time: 0.3212582 seconds

Benchmarking for Reverse Ordered array:
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Reverse Ordered (n=100) with 5 runs
  n=100, Time: 0.590375 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Reverse Ordered (n=200) with 5 runs
  n=200, Time: 0.23020000000000002 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Reverse Ordered (n=400) with 5 runs
  n=400, Time: 0.2502 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Reverse Ordered (n=800) with 5 runs
  n=800, Time: 0.28985 seconds
2024-02-05 16:36:21 INFO  Benchmark_Timer - Begin run: Reverse Ordered (n=1600) with 5 runs
  n=1600, Time: 0.2827332 seconds
```

Fig: Output of Partially and Reverse ordered array


InsertionSortTest (edu.neu.coe. 279 ms
testMutatingInsertionSort 233 ms
sort0 22 ms
sort1 14 ms
sort2 5 ms
sort3 3 ms
testStaticInsertionSort 2 ms

```java
import ...

/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",  cutoff: "",  interimInvers
        Helper<Integer> helper = HelperFactory.create( description: "InsertionSort", list.size(), config);
```

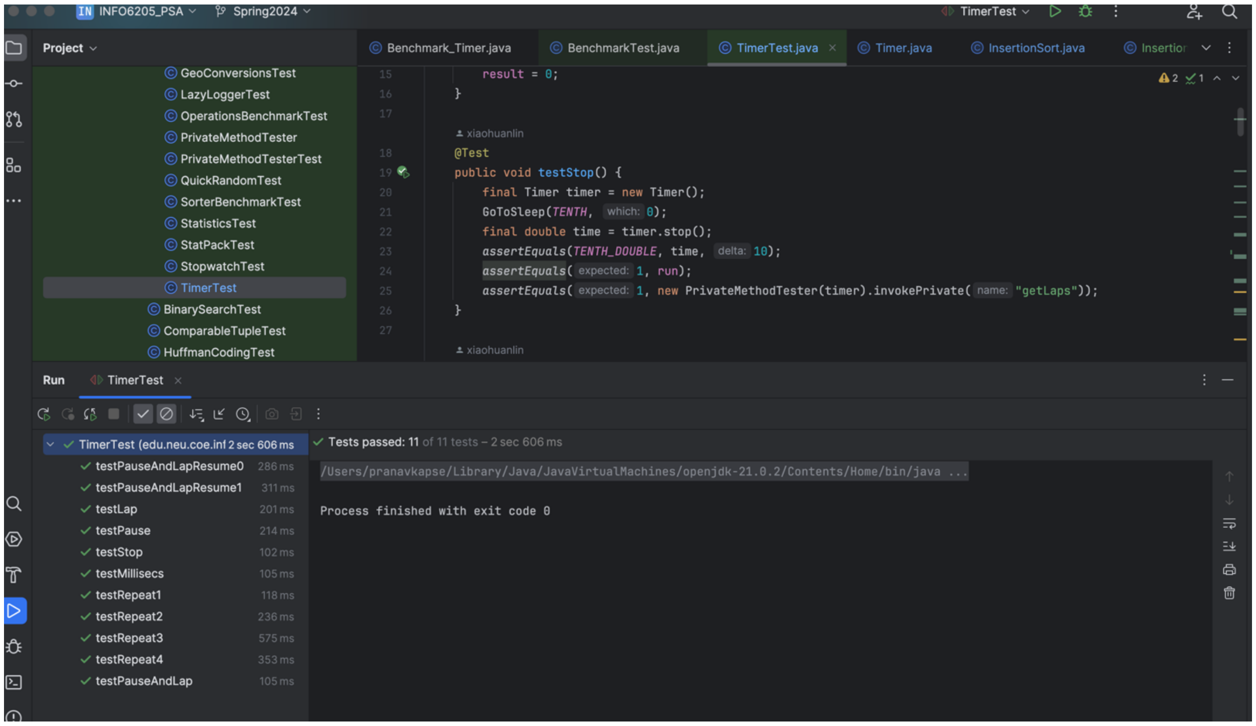
```
Tests passed: 6 of 6 tests – 279 ms
/Users/pranavkapse/Library/Java/JavaVirtualMachines/openjdk-21.0.2/Contents/Home/bin/java ...
Helper for InsertionSort with 4 elements
StatPack {hits: 9,880, normalized=21.454; copies: 0, normalized=0.000; inversions: 2,421, normalized=5.257; swaps: 2,421, norm
StatPack {hits: 19,800, normalized=42.995; copies: 0, normalized=0.000; inversions: 4,950, normalized=10.749; swaps: 4,950, no

Process finished with exit code 0
```

Fig: Insertion Sort passed all Test cases

Timer Class methods:

1. ‘repeat’ method:
    - Using this method, a function or functions are run for a certain number of repetitions, and the time required for each repetition is recorded.
    - It makes use of a Function to carry out the computation itself, a Supplier to obtain input values (if any), and optional pre- and post-processing functions.
    - It computes the average time per repetition by timing how long each repetition takes.
    - Additionally, it manages a warm-up phase in which certain initial runs are not included in the computation.
    - The average milliseconds per repetition is the outcome.
2. ‘clock’ method:
    - The current system time is obtained using this method in nanoseconds.
    - It has been used to calculate how long it takes to perform repeated function calls.
    - To keep the toMillisecs method consistent, the returned value is expressed in nanoseconds.

3. <u>‘toMillisecs’ method:</u>
    - Using this method, one can convert a given nanosecond time interval to milliseconds.
    - It is employed to present the elapsed time in a way that is easier for humans to understand.
    - To convert nanoseconds to milliseconds, divide the input value by 1,000,000.0 (1 million).

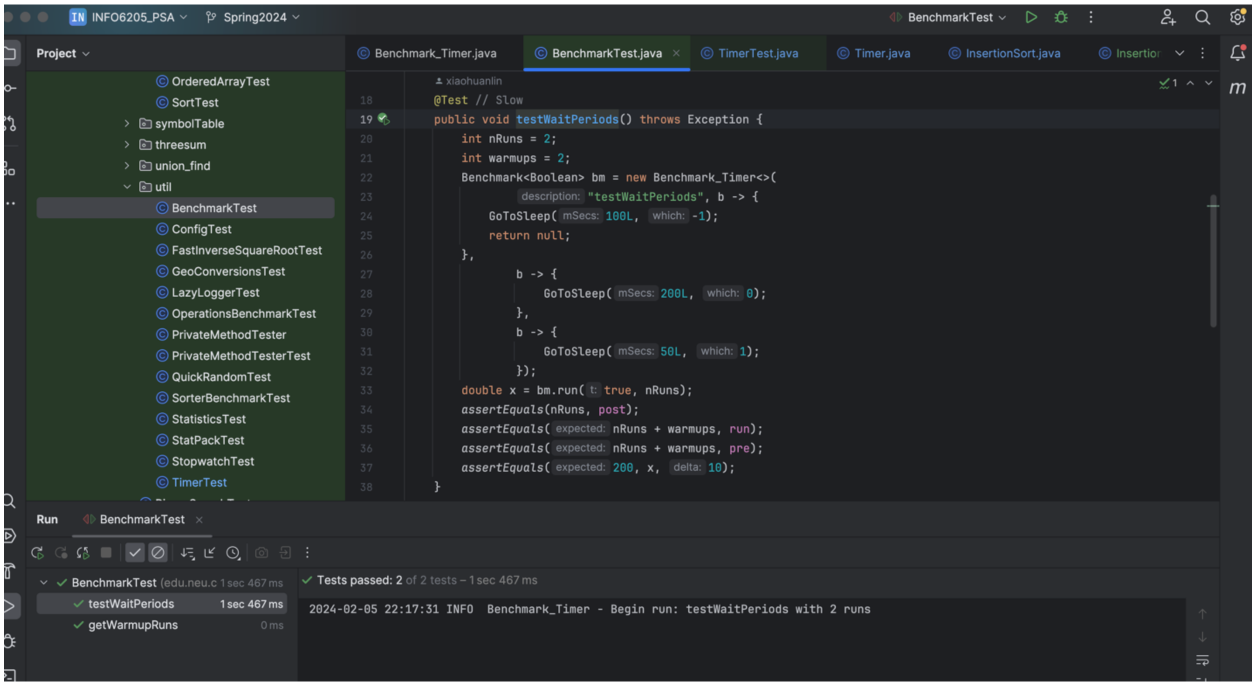


```java
        result = 0;
    }

    @Test
    public void testStop() {
        final Timer timer = new Timer();
        GoToSleep(TENTH, 0);
        final double time = timer.stop();
        assertEquals(TENTH_DOUBLE, time, 10);
        assertEquals(1, run);
        assertEquals(1, new PrivateMethodTester(timer).invokePrivate("getLaps"));
    }
```

```
TimerTest (edu.neu.coe.inf 2 sec 606 ms
  testPauseAndLapResume0   286 ms
  testPauseAndLapResume1   311 ms
  testLap                  201 ms
  testPause                214 ms
  testStop                 102 ms
  testMillisecs            105 ms
  testRepeat1              118 ms
  testRepeat2              236 ms
  testRepeat3              575 ms
  testRepeat4              353 ms
  testPauseAndLap          105 ms
```

```
Tests passed: 11 of 11 tests – 2 sec 606 ms
/Users/pranavkapse/Library/Java/JavaVirtualMachines/openjdk-21.0.2/Contents/Home/bin/java ...

Process finished with exit code 0
```

Fig: All test cases of TimerTest class passed



```java
@Test // Slow
public void testWaitPeriods() throws Exception {
    int nRuns = 2;
    int warmups = 2;
    Benchmark<Boolean> bm = new Benchmark_Timer<>(
            description: "testWaitPeriods", b -> {
        GoToSleep( mSecs: 100L,  which: -1);
        return null;
    },
            b -> {
                GoToSleep( mSecs: 200L,  which: 0);
            },
            b -> {
                GoToSleep( mSecs: 50L,  which: 1);
            });
    double x = bm.run( t: true, nRuns);
    assertEquals(nRuns, post);
    assertEquals( expected: nRuns + warmups, run);
    assertEquals( expected: nRuns + warmups, pre);
    assertEquals( expected: 200, x,  delta: 10);
}
```

```
BenchmarkTest (edu.neu.c 1 sec 467 ms
  testWaitPeriods        1 sec 467 ms
  getWarmupRuns                  0 ms

Tests passed: 2 of 2 tests – 1 sec 467 ms

2024-02-05 22:17:31 INFO  Benchmark_Timer - Begin run: testWaitPeriods with 2 runs
```

Fig: All test cases of BenchmarkTest class passed