# Assignment 3

- (Part 1) You are to 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. The APIs of these class are as follows:

```
public interface Benchmark<T> {
   default double run(T t, int m) {
      return runFromSupplier(() -> t, m);
   }

   double runFromSupplier(Supplier<T> supplier, int m);
}
```

[Supplier is a Java function type which supplies values of type T using the method: get().]

```
public class Benchmark_Timer<T> implements Benchmark<T> {
```

```
public Benchmark_Timer(String description, UnaryOperator<T> fPre, Consumer<T> fRun, Consumer<T> fPost)
```

[Consumer<T> is a Java function type which consumes a type T with the method: accept(t).

UnaryOperator<T> is essentially an alias of Function<T, T> which defines apply(t) which takes a T and returns a T.]

```
public Benchmark_Timer(String description, UnaryOperator<T> fPre, Consumer<T> fRun)
```

```
public Benchmark_Timer(String description, Consumer<T> fRun, Consumer<T> fPost)
```

```
public Benchmark_Timer(String description, Consumer<T> f)
```

```
public class Timer {
... // see below for methods to be implemented...
}
```

```
public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> function, UnaryOperator<T> preFunction, Consumer<U> postFunction) {
// TO BE IMPLEMENTED
}
```

[Function<T, U> defines a method U apply(t: T), which takes a value of T and returns a value of U.]

```
private static long getClock() {
   // TO BE IMPLEMENTED
}private static double toMillisecs(long ticks) {
   // TO BE IMPLEMENTED
}
```

The function to be timed, hereinafter the "target" function, is the Consumer function fRun (or just f) passed in to one or other of the constructors. For example, you might create a function which sorts an array with n elements.

The generic type T is that of the input to the target function.

The first parameter to the first run method signature is the parameter that will, in turn, be passed to target function. In the second signature, supplier will be invoked each time to get a t which is passed to the other run method.

The second parameter to the run function (m) is the number of times the target function will be called.

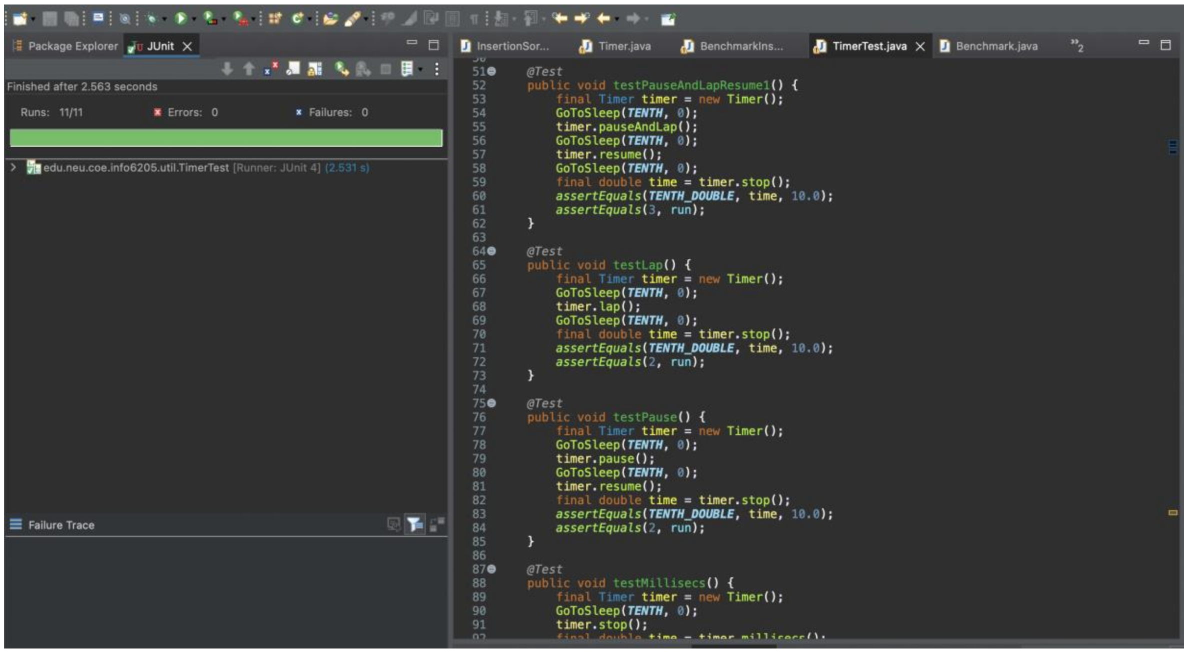
The return value from run is the average number of milliseconds taken for each run of the target function.

- (Part 2) Implement InsertionSort (in the InsertionSort class) by simply looking up the insertion code used by Arrays.sort. If you have the instrument = true setting in test/resources/config.ini, then you will need to use the helper methods for comparing and swapping (so that they properly count the number of swaps/compares). 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.

- (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. 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.

## Unit Test cases:

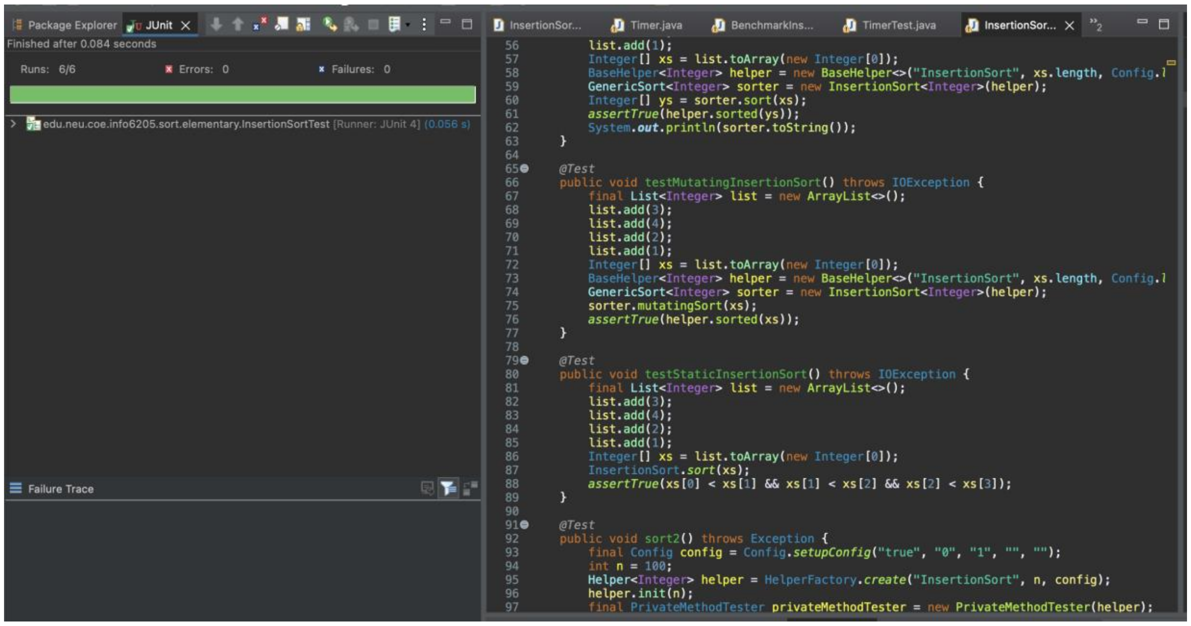
### TimerTest.java



```
Finished after 2.563 seconds

Runs: 11/11        Errors: 0        Failures: 0

> edu.neu.coe.info6205.util.TimerTest [Runner: JUnit 4] (2.531 s)
```

```java
    @Test
    public void testPauseAndLapResume1() {
        final Timer timer = new Timer();
        GoToSleep(TENTH, 0);
        timer.pauseAndLap();
        GoToSleep(TENTH, 0);
        timer.resume();
        GoToSleep(TENTH, 0);
        final double time = timer.stop();
        assertEquals(TENTH_DOUBLE, time, 10.0);
        assertEquals(3, run);
    }

    @Test
    public void testLap() {
        final Timer timer = new Timer();
        GoToSleep(TENTH, 0);
        timer.lap();
        GoToSleep(TENTH, 0);
        final double time = timer.stop();
        assertEquals(TENTH_DOUBLE, time, 10.0);
        assertEquals(2, run);
    }

    @Test
    public void testPause() {
        final Timer timer = new Timer();
        GoToSleep(TENTH, 0);
        timer.pause();
        GoToSleep(TENTH, 0);
        timer.resume();
        final double time = timer.stop();
        assertEquals(TENTH_DOUBLE, time, 10.0);
        assertEquals(2, run);
    }

    @Test
    public void testMillisecs() {
        final Timer timer = new Timer();
        GoToSleep(TENTH, 0);
        timer.stop();
        final double time = timer.millisecs();
```

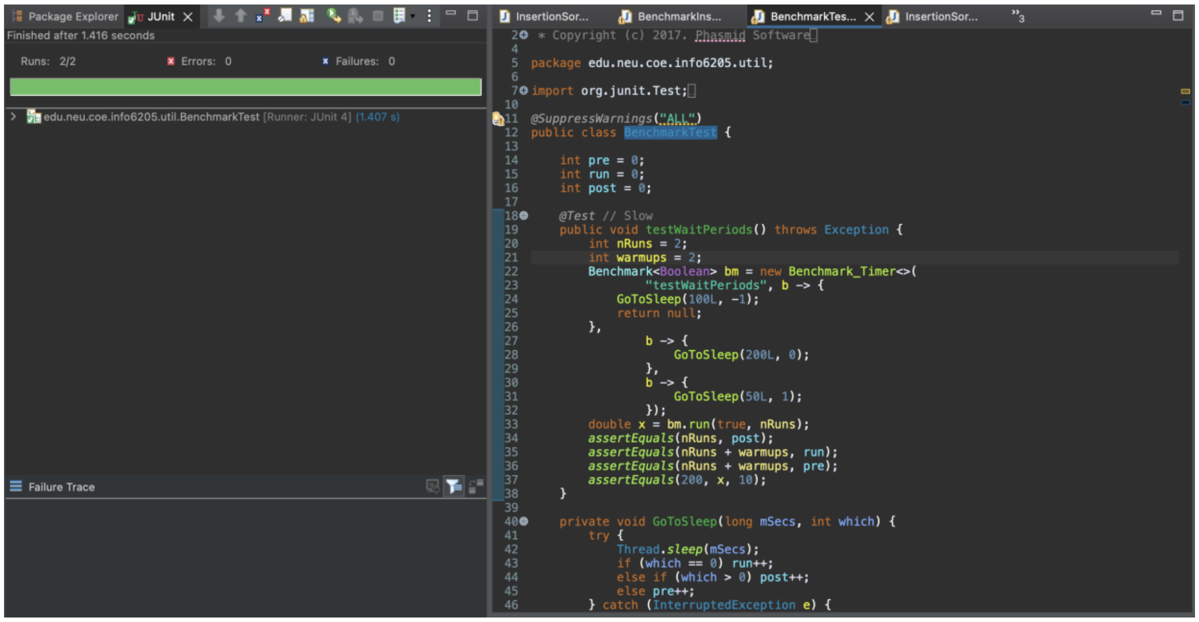
### InsertionSortTest.java



```
Finished after 0.084 seconds

Runs: 6/6        Errors: 0        Failures: 0

> edu.neu.coe.info6205.sort.elementary.InsertionSortTest [Runner: JUnit 4] (0.056 s)
```

```java
        list.add(1);
        Integer[] xs = list.toArray(new Integer[0]);
        BaseHelper<Integer> helper = new BaseHelper<>("InsertionSort", xs.length, Config.l
        GenericSort<Integer> sorter = new InsertionSort<Integer>(helper);
        Integer[] ys = sorter.sort(xs);
        assertTrue(helper.sorted(ys));
        System.out.println(sorter.toString());
    }

    @Test
    public void testMutatingInsertionSort() throws IOException {
        final List<Integer> list = new ArrayList<>();
        list.add(3);
        list.add(4);
        list.add(2);
        list.add(1);
        Integer[] xs = list.toArray(new Integer[0]);
        BaseHelper<Integer> helper = new BaseHelper<>("InsertionSort", xs.length, Config.l
        GenericSort<Integer> sorter = new InsertionSort<Integer>(helper);
        sorter.mutatingSort(xs);
        assertTrue(helper.sorted(xs));
    }

    @Test
    public void testStaticInsertionSort() throws IOException {
        final List<Integer> list = new ArrayList<>();
        list.add(3);
        list.add(4);
        list.add(2);
        list.add(1);
        Integer[] xs = list.toArray(new Integer[0]);
        InsertionSort.sort(xs);
        assertTrue(xs[0] < xs[1] && xs[1] < xs[2] && xs[2] < xs[3]);
    }

    @Test
    public void sort2() throws Exception {
        final Config config = Config.setupConfig("true", "0", "1", "", "");
        int n = 100;
        Helper<Integer> helper = HelperFactory.create("InsertionSort", n, config);
        helper.init(n);
        final PrivateMethodTester privateMethodTester = new PrivateMethodTester(helper);
```

### BenchmarkTest.java



```
Finished after 1.416 seconds

Runs: 2/2        Errors: 0        Failures: 0

edu.neu.coe.info6205.util.BenchmarkTest [Runner: JUnit 4] (1.407 s)
```

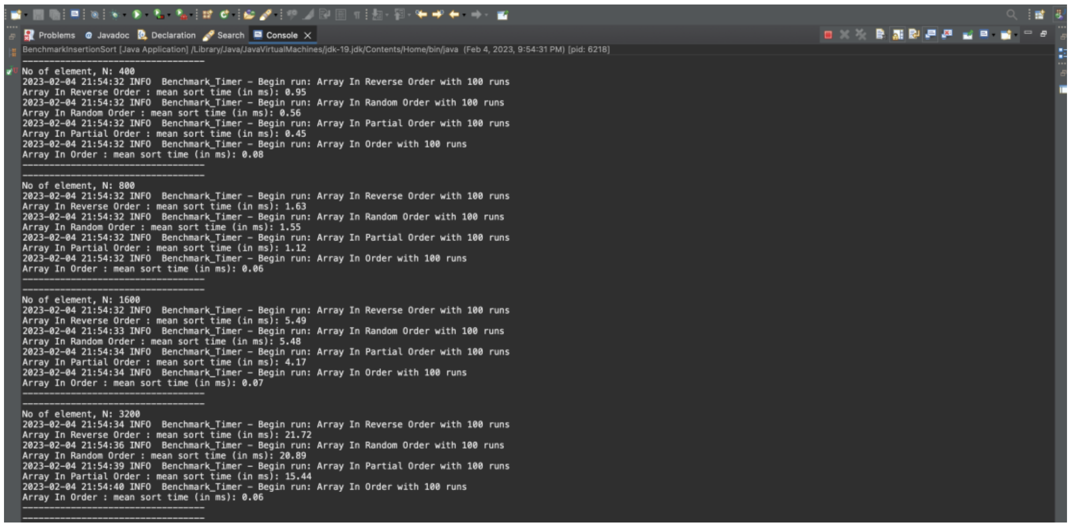
```java
 * Copyright (c) 2017. Phasmid Software

package edu.neu.coe.info6205.util;

import org.junit.Test;

@SuppressWarnings("ALL")
public class BenchmarkTest {

    int pre = 0;
    int run = 0;
    int post = 0;

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

    private void GoToSleep(long mSecs, int which) {
        try {
            Thread.sleep(mSecs);
            if (which == 0) run++;
            else if (which > 0) post++;
            else pre++;
        } catch (InterruptedException e) {
```

## Observation and Evidence:

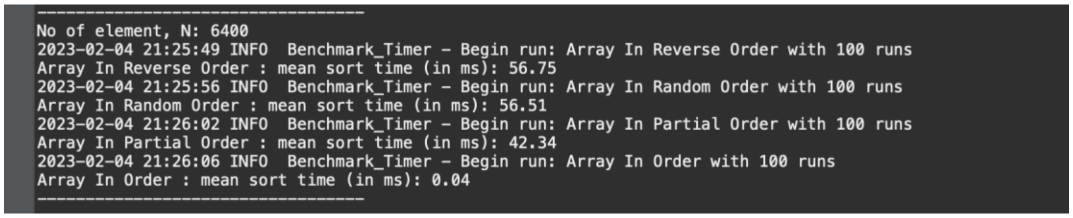
The reverse-sorted array requires the longest amount of time to complete the insertion sort algorithm due to the need for numerous number swaps. This correlation is demonstrated in the benchmarking results shown in the observation table. The table displays that, among the various types of arrays, the reverse-sorted array takes the longest to run the insertion sort, followed by the randomly sorted array, which is faster than the reverse-sorted array. The partially sorted array takes less time than the randomly sorted array, and the sorted array takes the shortest amount of time to run the insertion sort.

Therefore, **Ordered < Partially Ordered < Randomly Ordered < Reverse Ordered.**

Screenshot of the observation:



```
BenchmarkInsertionSort [Java Application] /Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bin/java  (Feb 4, 2023, 9:54:31 PM) [pid: 6218]
----------------------------------
No of element, N: 400
2023-02-04 21:54:32 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
Array In Reverse Order : mean sort time (in ms): 0.95
2023-02-04 21:54:32 INFO  Benchmark_Timer - Begin run: Array In Random Order with 100 runs
Array In Random Order : mean sort time (in ms): 0.56
2023-02-04 21:54:32 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 0.45
2023-02-04 21:54:32 INFO  Benchmark_Timer - Begin run: Array In Order with 100 runs
Array In Order : mean sort time (in ms): 0.08
----------------------------------
----------------------------------
No of element, N: 800
2023-02-04 21:54:32 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
Array In Reverse Order : mean sort time (in ms): 1.63
2023-02-04 21:54:32 INFO  Benchmark_Timer - Begin run: Array In Random Order with 100 runs
Array In Random Order : mean sort time (in ms): 1.55
2023-02-04 21:54:32 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 1.12
2023-02-04 21:54:32 INFO  Benchmark_Timer - Begin run: Array In Order with 100 runs
Array In Order : mean sort time (in ms): 0.06
----------------------------------
----------------------------------
No of element, N: 1600
2023-02-04 21:54:32 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
Array In Reverse Order : mean sort time (in ms): 5.49
2023-02-04 21:54:33 INFO  Benchmark_Timer - Begin run: Array In Random Order with 100 runs
Array In Random Order : mean sort time (in ms): 5.48
2023-02-04 21:54:34 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 4.17
2023-02-04 21:54:34 INFO  Benchmark_Timer - Begin run: Array In Order with 100 runs
Array In Order : mean sort time (in ms): 0.07
----------------------------------
----------------------------------
No of element, N: 3200
2023-02-04 21:54:34 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
Array In Reverse Order : mean sort time (in ms): 21.72
2023-02-04 21:54:36 INFO  Benchmark_Timer - Begin run: Array In Random Order with 100 runs
Array In Random Order : mean sort time (in ms): 20.89
2023-02-04 21:54:39 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 15.44
2023-02-04 21:54:40 INFO  Benchmark_Timer - Begin run: Array In Order with 100 runs
Array In Order : mean sort time (in ms): 0.06
----------------------------------
----------------------------------
```

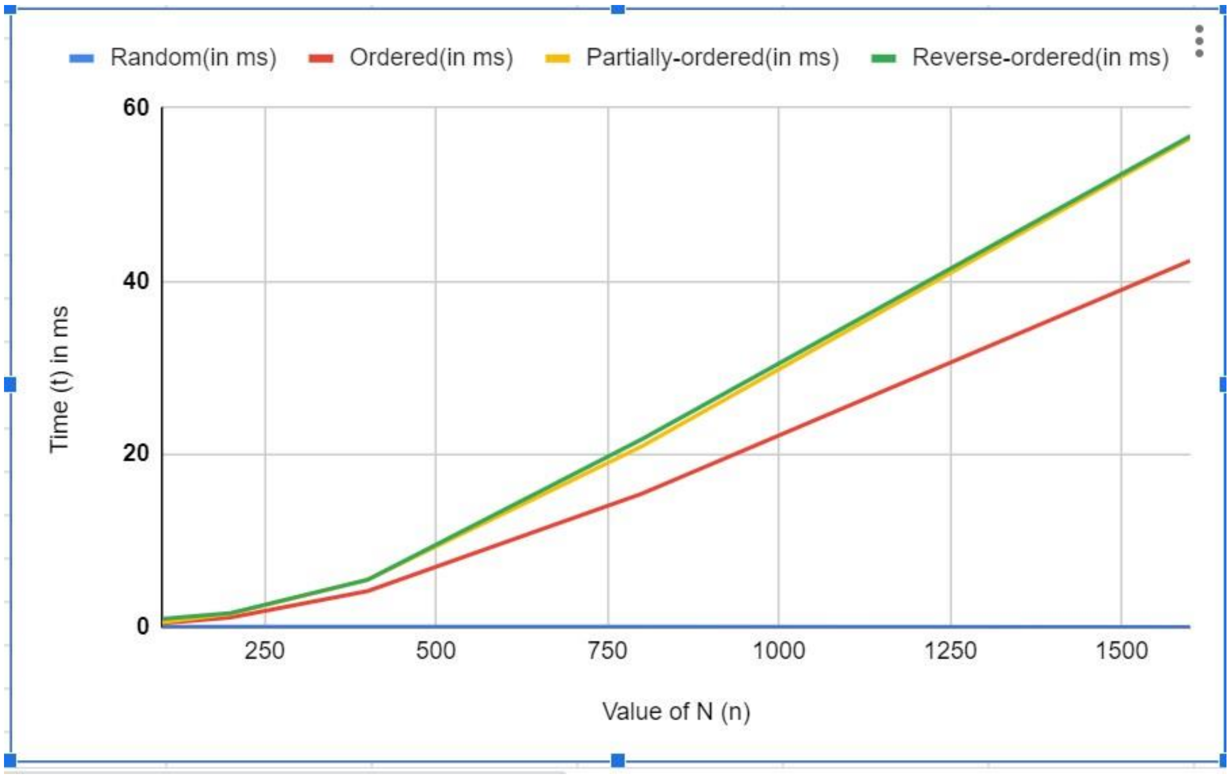


```
----------------------------------
No of element, N: 6400
2023-02-04 21:25:49 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
Array In Reverse Order : mean sort time (in ms): 56.75
2023-02-04 21:25:56 INFO  Benchmark_Timer - Begin run: Array In Random Order with 100 runs
Array In Random Order : mean sort time (in ms): 56.51
2023-02-04 21:26:02 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 42.34
2023-02-04 21:26:06 INFO  Benchmark_Timer - Begin run: Array In Order with 100 runs
Array In Order : mean sort time (in ms): 0.04
----------------------------------
```

## Raw time data observation:

| Value of N | Random(in ms) | Ordered(in ms) | Partially-ordered(in ms) | Reverse-ordered(in ms) |
|---|---|---|---|---|
| 100 | 0.08 | 0.45 | 0.56 | 0.95 |
| 200 | 0.06 | 1.12 | 1.55 | 1.63 |
| 400 | 0.07 | 4.17 | 5.48 | 5.49 |
| 800 | 0.06 | 15.44 | 20.89 | 21.72 |
| 1600 | 0.04 | 42.34 | 56.51 | 56.75 |

Plotted Graph from the raw timing observations of the above table:



**Conclusion:**

The data in the chart provides information about the running time of an algorithm with various types of input arrays. The best case scenario is when the input array is sequentially ordered, and it takes the shortest amount of time to run. On the other hand, the worst case scenario is when the input array is randomly ordered, and it takes the longest amount of time to run. The average case scenario is calculated by taking the average of the running time for all four types of arrays. As the size of the input array increases, the average running time has a growth rate that can be described as linearithmic.