

Program Structures and Algorithms
Spring 2023(Section - 1)

Name: Mani Charan Reddy Loka

NUID: 002727403

Task:

(Part 1) You are to implement three (3) methods (*repeat*, *getClock*, and *toMillisecs*) of a class called *Timer*.

Repeat method:

```
public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> function, UnaryOperator<U> postFunction) {
    logger.trace("repeat: with " + n + " runs");

    ticks=0;
    running=false;

    for (int i = 0; i < n; i++) {
        T t = supplier.get();
        if (preFunction != null) {
            t = preFunction.apply(t);
        }
        resume();
        U u = function.apply(t);
        pauseAndLap();
        if (postFunction != null) {
            postFunction.accept(u);
        }
    }
    final double result = meanLapTime();
    resume();
    return result;
} // FIXME: note that the timer is running when this method is called and should still be running
```

getClock method:

```
private static long getClock() {
    // FIXME by replacing the following code
    return System.nanoTime();
    // END
}
```

toMillisecs method:

```
private static double toMillisecs(long ticks) {
    // FIXME by replacing the following code
    //System.out.println(ticks);
    return ticks/1000000.0;
    // END
}
```

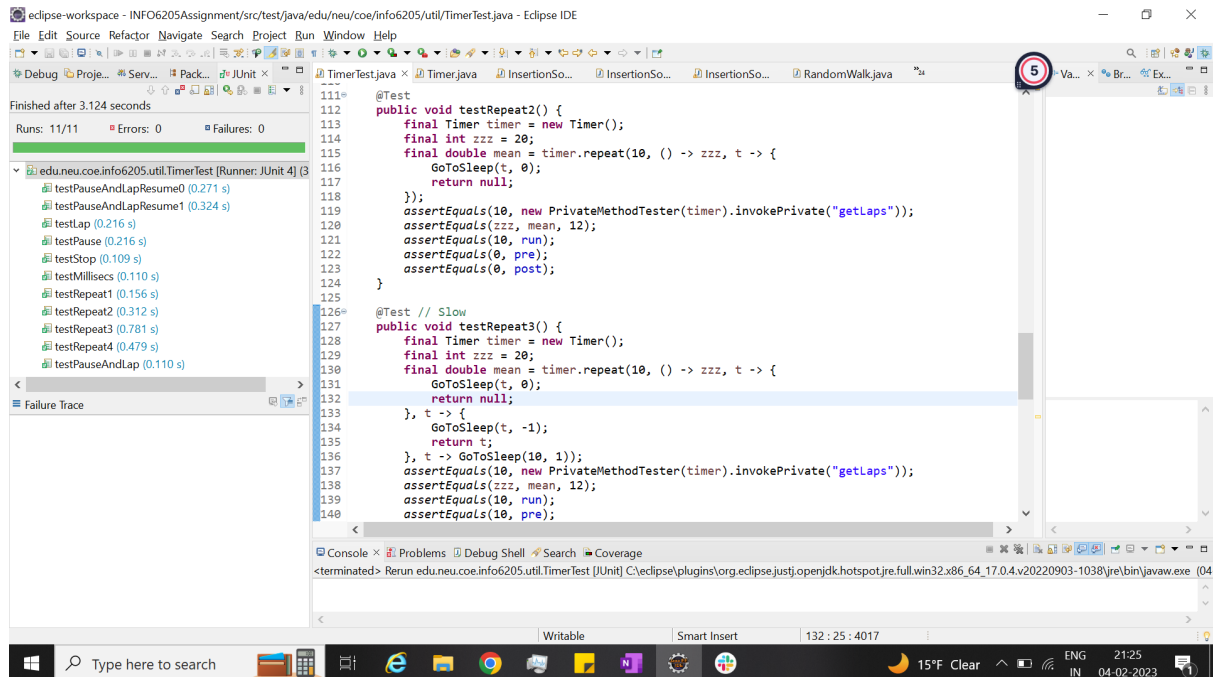
Program Structures and Algorithms

Spring 2023(Section - 1)

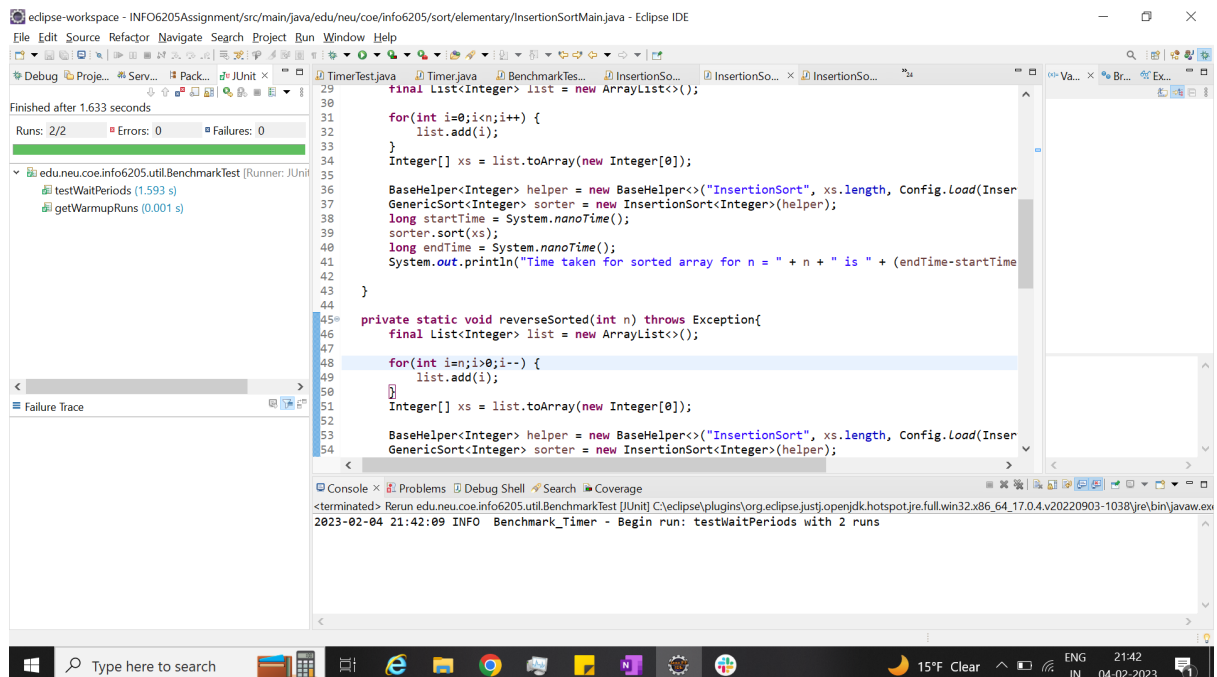
Unit Test Screenshots:

1) TimerTest.java

Here, I updated the delta to a slightly higher value of 12 because I was always getting a deviation of around 10~11.



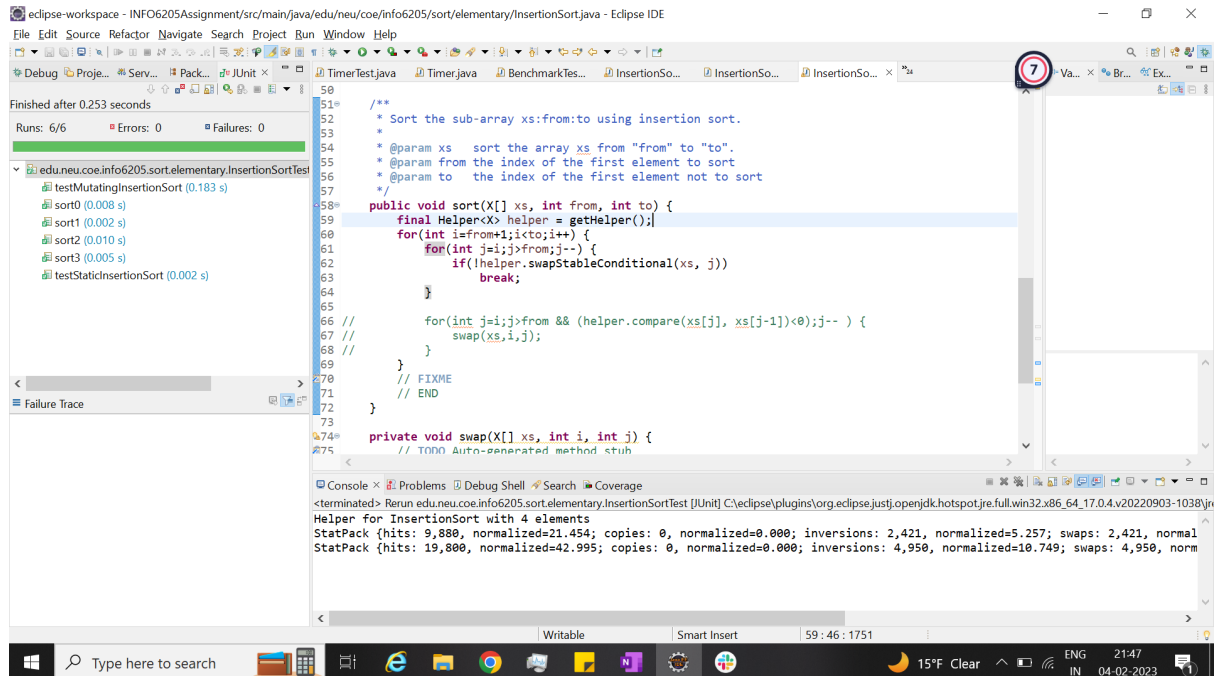
2) BenchmarkTest.java



Program Structures and Algorithms Spring 2023(Section - 1)

(Part 2) Implement *InsertionSort* (in the *InsertionSort* class)

Code:



The screenshot shows the Eclipse IDE with the `InsertionSort.java` file open. The code implements a sorting algorithm. The left sidebar shows the project structure with `edu.neu.coe.info6205.sort.elementary.InsertionSortTest` selected. The bottom console shows the output of running the tests.

```
50  
51  
52 /**  
53  * Sort the sub-array xs:from:to using insertion sort.  
54  * @param xs  sort the array xs from "from" to "to".  
55  * @param from the index of the first element to sort  
56  * @param to   the index of the first element not to sort  
57  */  
58  
59 public void sort(X[] xs, int from, int to) {  
60     final Helper<X> helper = getHelper();  
61     for(int i=from+1; i<to; i++) {  
62         for(int j=i; j>from; j--) {  
63             if(!helper.swapStableConditional(xs, j))  
64                 break;  
65  
66             for(int j=i; j>from && (helper.compare(xs[j], xs[j-1])<0); j--) {  
67                 swap(xs, j, j-1);  
68             }  
69         }  
70     }  
71 }  
72  
73  
74 private void swap(X[] xs, int i, int j) {  
75     // TODO Auto-generated method stub
```

Unit Test Results:

- testMutatingInsertionSort (0.183 s)
- sort0 (0.008 s)
- sort1 (0.002 s)
- sort2 (0.010 s)
- sort3 (0.005 s)
- testStaticInsertionSort (0.002 s)

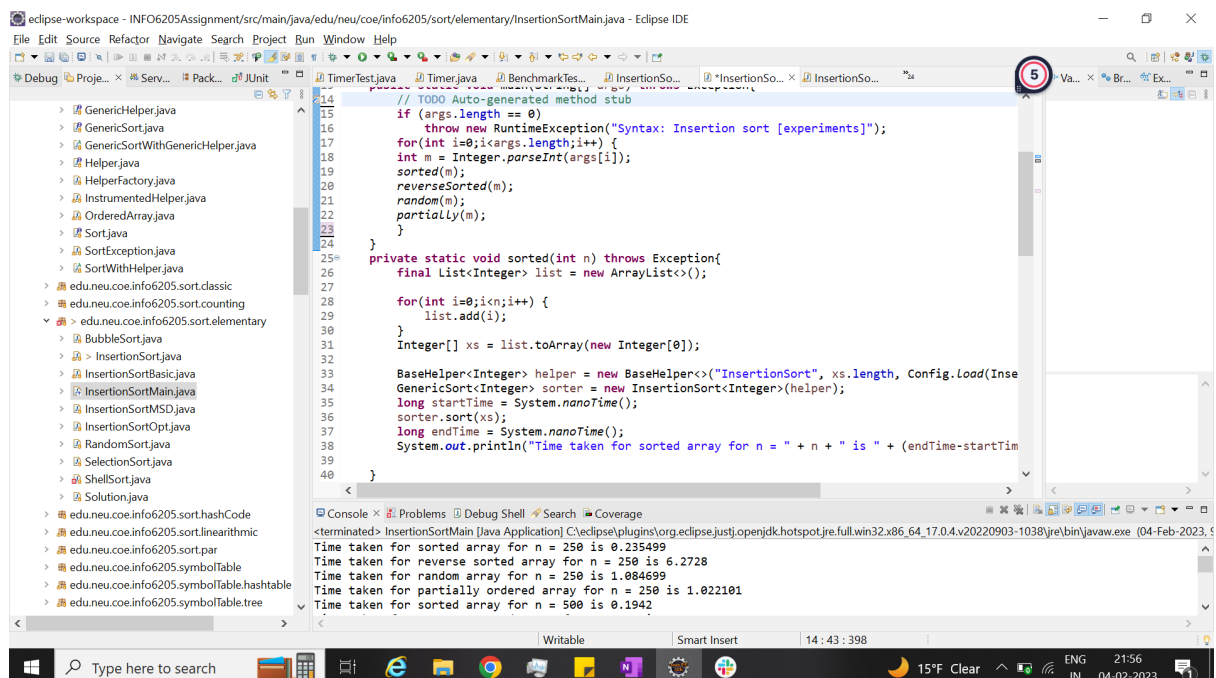
Console Output:

```
<terminated> Rerun edu.neu.coe.info6205.sort.elementary.InsertionSortTest [JUnit] C:\eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.17.0.4.v20220903-1038\jre\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; inversions: 4,950, normalized=10.749; swaps: 4,950, normalized=10.749}
```

Unit Test Screenshot:

(Part 3) Implement a main program (or you could do it via your own unit tests) to actually run the following benchmarks:

Main method:



The screenshot shows the Eclipse IDE with the `InsertionSortMain.java` file open. The code implements a main method that runs benchmarks. The left sidebar shows the project structure with `edu.neu.coe.info6205.sort.elementary.InsertionSortMain` selected. The bottom console shows the output of running the benchmarks.

```
14  
15 // TODO Auto-generated method stub  
16 if (args.length == 0) {  
17     throw new RuntimeException("Syntax: Insertion sort [experiments]");  
18 }  
19 for(int i=0; i<args.length; i++) {  
20     int m = Integer.parseInt(args[i]);  
21     sorted(m);  
22     reverseSorted(m);  
23     random(m);  
24     partially(m);  
25 }  
26  
27 private static void sorted(int n) throws Exception {  
28     final List<Integer> list = new ArrayList<>();  
29     for(int i=0; i<n; i++) {  
30         list.add(i);  
31     }  
32     Integer[] xs = list.toArray(new Integer[0]);  
33  
34     BaseHelper<Integer> helper = new BaseHelper<>("InsertionSort", xs.length, Config.Load(Inse  
35     GenericSort<Integer> sorter = new InsertionSort<Integer>(helper);  
36     long startTime = System.nanoTime();  
37     sorter.sort(xs);  
38     long endTime = System.nanoTime();  
39     System.out.println("Time taken for sorted array for n = " + n + " is " + (endTime-startTim  
40 }
```

Console Output:

```
<terminated> InsertionSortMain [Java Application] C:\eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.17.0.4.v20220903-1038\jre\bin\javaw.exe (04-Feb-2023, 14:43:39)  
Time taken for sorted array for n = 250 is 0.235499  
Time taken for reverse sorted array for n = 250 is 6.2728  
Time taken for random array for n = 250 is 1.084699  
Time taken for partially ordered array for n = 250 is 1.022101  
Time taken for sorted array for n = 500 is 0.1942
```

Program Structures and Algorithms

Spring 2023(Section - 1)

Output:

```

// TODO Auto-generated method stub
if (args.length == 0)
    throw new RuntimeException("Syntax: Insertion sort [experiments]");
for(int i=0;i<args.length;i++) {

```

```

<terminated> InsertionSortMain [Java Application] C:\eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.17.0.4.v20220903-1038\jre\bin\javaw.exe (04-Feb-2023, 5
Time taken for sorted array for n = 250 is 0.235499
Time taken for reverse sorted array for n = 250 is 6.2728
Time taken for random array for n = 250 is 1.084699
Time taken for partially ordered array for n = 250 is 1.022101
Time taken for sorted array for n = 500 is 0.1942
Time taken for reverse sorted array for n = 500 is 9.724599
Time taken for random array for n = 500 is 7.937001
Time taken for partially ordered array for n = 500 is 0.770499
Time taken for sorted array for n = 1000 is 0.0453
Time taken for reverse sorted array for n = 1000 is 2.8742
Time taken for random array for n = 1000 is 2.7221
Time taken for partially ordered array for n = 1000 is 2.0948
Time taken for sorted array for n = 2000 is 0.036499
Time taken for reverse sorted array for n = 2000 is 16.149401
Time taken for random array for n = 2000 is 6.3544
Time taken for partially ordered array for n = 2000 is 4.3951
Time taken for sorted array for n = 4000 is 0.0449
Time taken for reverse sorted array for n = 4000 is 46.199999
Time taken for random array for n = 4000 is 23.2193
Time taken for partially ordered array for n = 4000 is 18.2721
Time taken for sorted array for n = 8000 is 0.081001
Time taken for reverse sorted array for n = 8000 is 241.088401
Time taken for random array for n = 8000 is 163.6889
Time taken for partially ordered array for n = 8000 is 84.9498
Time taken for sorted array for n = 16000 is 0.1461
Time taken for reverse sorted array for n = 16000 is 886.0826
Time taken for random array for n = 16000 is 409.5759
Time taken for partially ordered array for n = 16000 is 422.1685

```

(b)Validating by using doubling benchmark:

Here, I timed the observations using the doubling method from n=250 until n=16000

N		Ordered		Partially		Random		Reverse	
		(millisec)	lg ratio	(millisec)	lg ratio	(millisec)	lg ratio	(millisec)	lg ratio
250	Raw Time	0.24		1.02		1.08		6.27	
500	Raw Time	0.19	-0.34	0.77	-0.41	7.94	2.88	9.72	0.63
1000	Raw Time	0.05	-1.93	2.09	1.44	2.72	-1.55	2.87	-1.76
2000	Raw Time	0.04	-0.32	4.4	1.07	6.35	1.22	16.15	2.49
4000	Raw Time	0.04	0	18.27	2.05	23.22	1.87	46.12	1.51
8000	Raw Time	0.08	1	84.95	2.22	163.69	2.82	241.09	2.39
16000	Raw Time	0.14	0.81	422.17	2.31	409.58	1.32	886.08	1.88

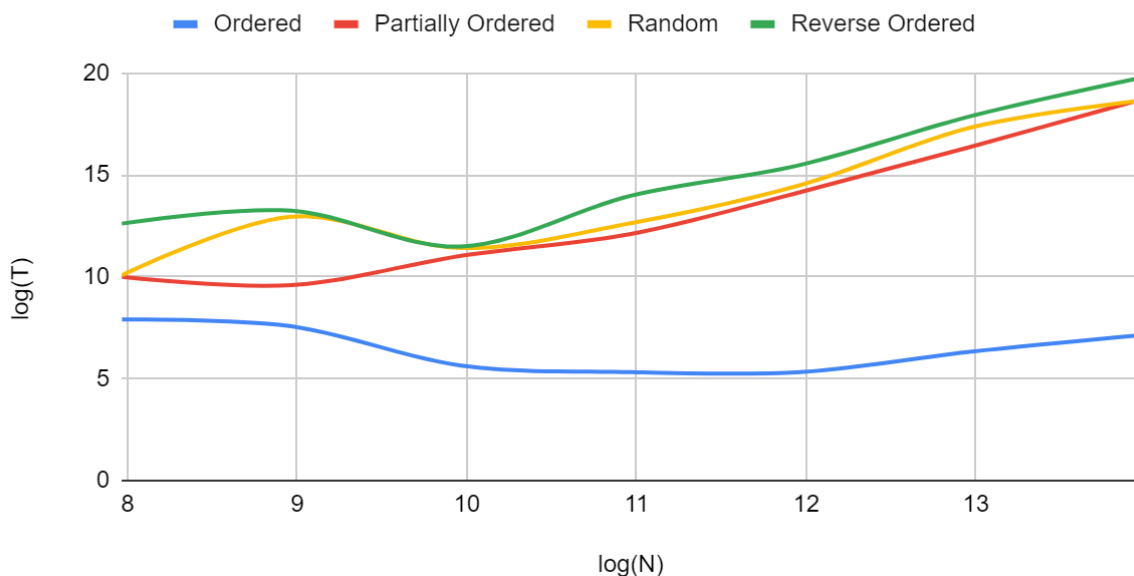
Program Structures and Algorithms
Spring 2023(Section - 1)

$\log(N)$ vs $\log(T)$ values for the above observations. Here, before applying the $\log(T)$, I multiplied every value with 10^3 in order to avoid negative values, as the log of something less than 1 is negative.

$\log(N)$	$\log(T)$ Ordered	$\log(T)$ Partially	$\log(T)$ Random	$\log(T)$ Reverse
7.965784285	7.906890596	9.994353437	10.0768156	12.61424973
8.965784285	7.569855608	9.588714636	12.95492329	13.2467406
9.965784285	5.64385619	11.02928723	11.40939094	11.48683502
10.96578428	5.321928095	12.10328781	12.63254088	13.97924654
11.96578428	5.321928095	14.15718901	14.50308035	15.49310489
12.96578428	6.321928095	16.37432633	17.32060666	17.87921229
13.96578428	7.129283017	18.68746454	18.64378574	19.75707743

Here's the graph of $\log(N)$ vs $\log(T)$ for four different initial array ordering situations: random, ordered, partially ordered, and reverse ordered

Ordered, Partially Ordered, Random, and Reverse Ordered vs $\log(N)$



Conclusion:

The order of growth is not very much impacted when the input array is sorted, while it increases with the size of the array in all other cases when the input array is partially sorted, random, or reverse ordered. It is always higher in the case of reverse ordered as it involves the highest number of swaps and a little lower for partially sorted and random array as it doesn't involve as many swaps as required by reverse ordered array.