

INFO 6205 – Spring 2022

Assignment 2

1. Task:

- Complete the implementation of Timer class and Benchmark_Timer class.
- Implement insertion sort in the InsertionSort class.
- Measure the running time of insertion sort in the following cases; Ordered array, Randomly ordered, Partially ordered array, and Reverse ordered array.

2. Output:

To produce the desired outputs, I coded the insertion sort to be timed for increasingly doubling elements (400, 800, 1600, 3200, 6200, 12800, 25600, 51200}. For each number, the time is averaged over 5 trails, and a warm- up session ran before running each trial.

Elements	Ordered	Partial	Random	Reverse
400	0.0ms	0.4ms	0.6ms	0.8ms
800	0.0ms	0.4ms	0.8ms	2.2ms
1600	0.0ms	0.8ms	4.0ms	7.8ms
3200	0.0ms	4.2ms	16.2ms	31.4ms
6400	0.0ms	18.0ms	72.0ms	142.4ms
12800	0.0ms	95.6ms	257.0ms	529.0ms
25600	0.1ms	333.4ms	1233.2ms	1903.6ms
51260	0.1ms	1337.2ms	5497.8ms	7771.2ms

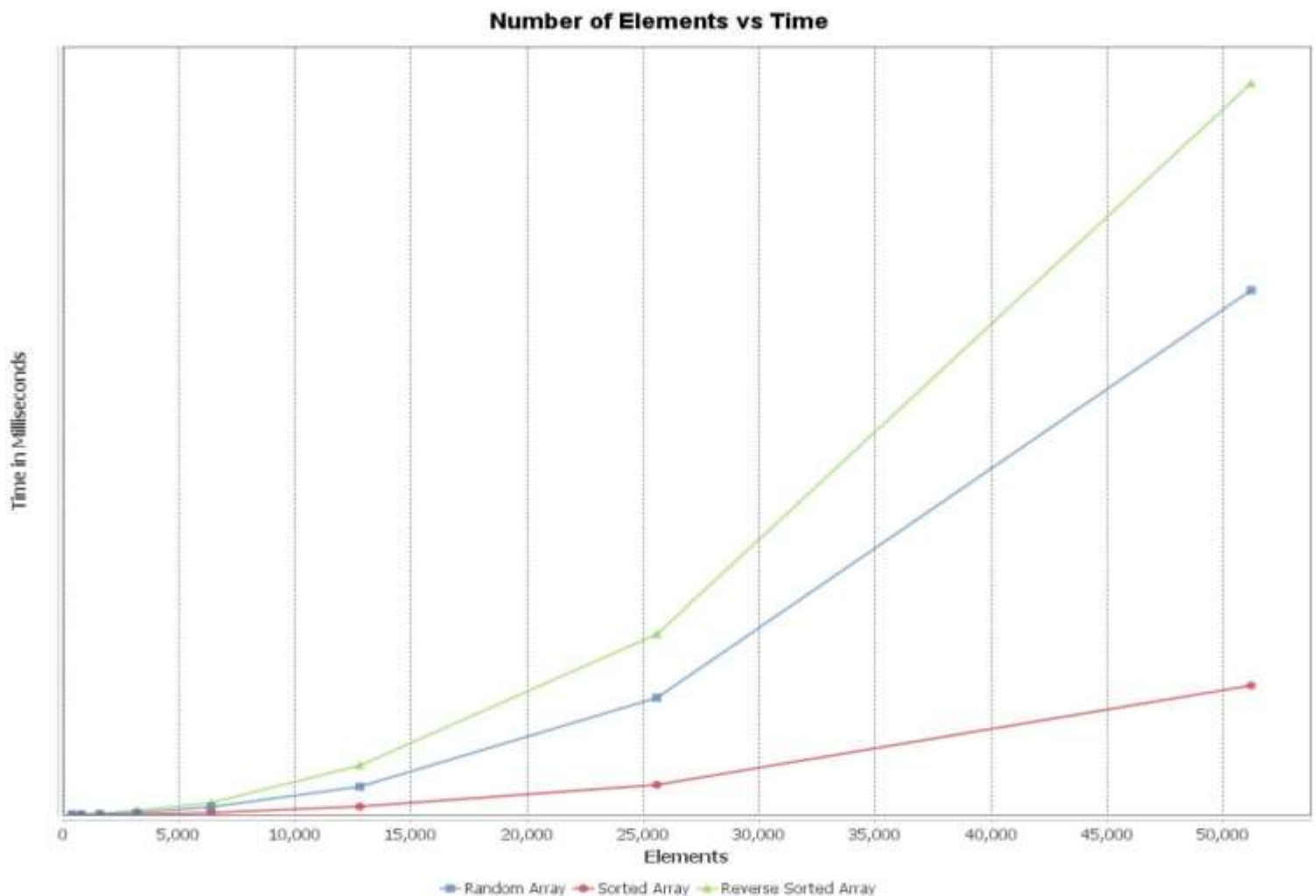
3. Observation:

We can learn a few things about insertion sort from the result displayed above.

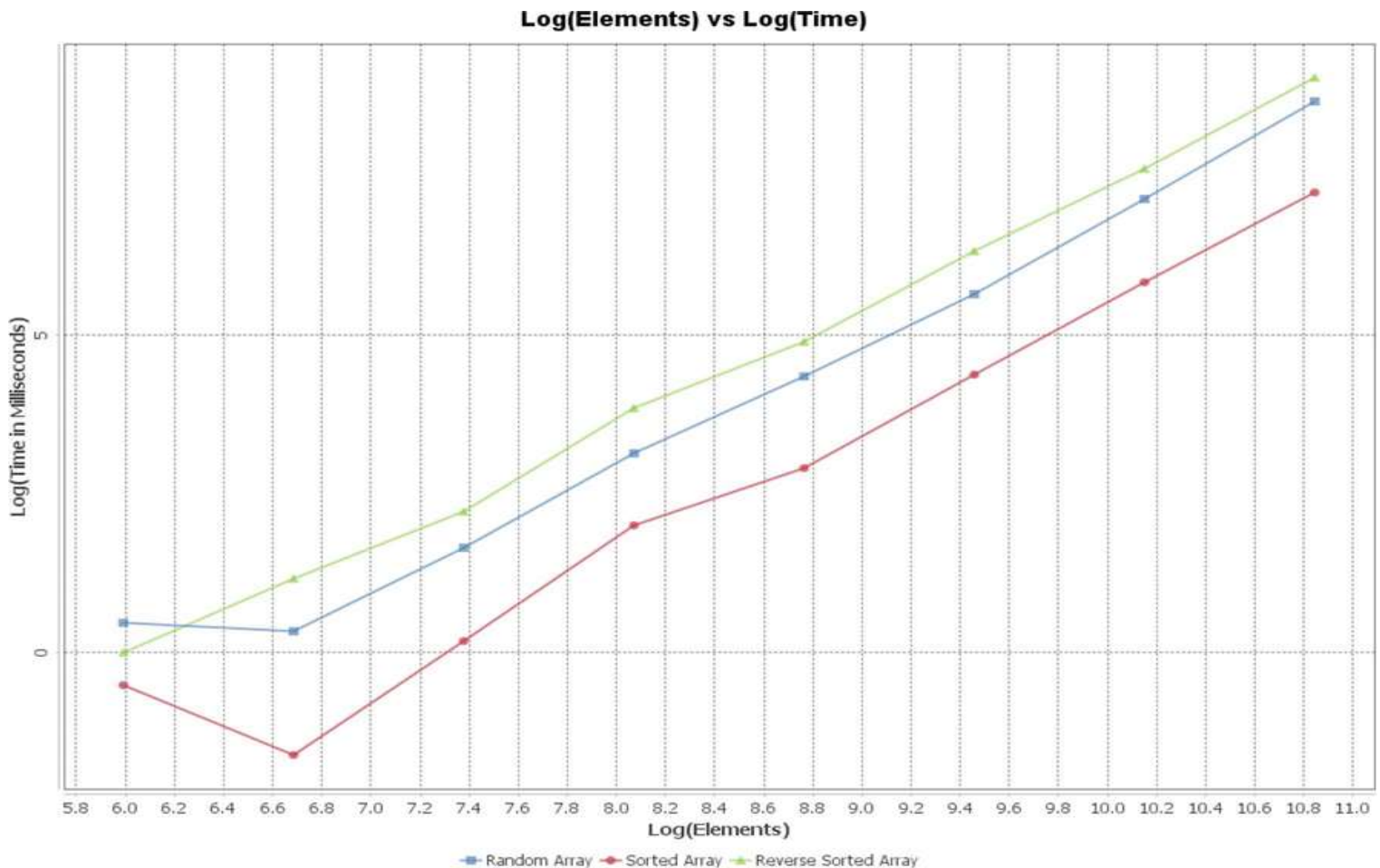
- As the doubling elements increase in number, we can see a quadratic increase in the time required by the procedure. Own* is the average time taken.
- For ordered or sorted arrays, insertion sort is the fastest. It is the best-case situation for insertion sort because the algorithm simply has to traverse the array without swapping. The time spent is nothing (n).

- The worst-case scenario is a reverse-ordered list, in which the algorithm performs numerous swaps for each and every entry. The time required is $O(n^2)$.
- Partially ordered lists are a better situation since the amount of swaps the algorithm needs execute is very modest, allowing the process to be faster. The time taken is $O(n)$.

The graphs below show the quadratic development of the algorithm's time and the difference between the Partially Ordered, Random, and Reverse Ordered Arrays.

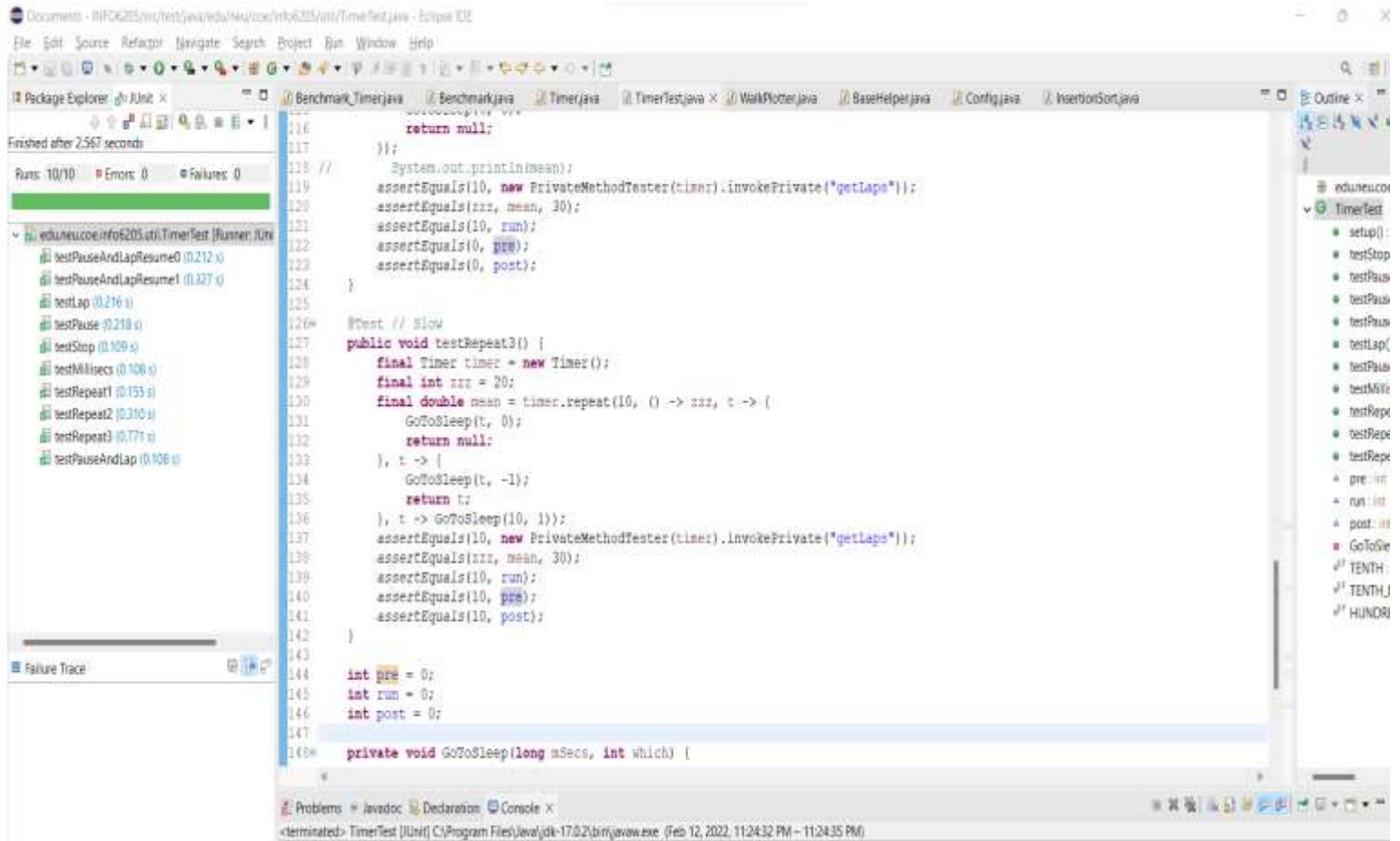


To prove a quadratic relationship between the number of elements in the array being sorted and time taken by the insertion sort algorithm, I plotted a graph of the logarithm of time taken against the logarithm



From the above graph, we can notice the graph following a straight- line trend, and hence we can state the time taken is a quadratic function of the number of elements. Hence proving the $O(n^2)$ average case time for insertion sort.

4. Unit Tests:



```

116         return null;
117     }
118     //
119     System.out.println(mean);
120     assertEquals(10, new PrivateMethodTester(timer).invokePrivate("getLaps"));
121     assertEquals(xxx, mean, 30);
122     assertEquals(10, run);
123     assertEquals(0, pre);
124     assertEquals(0, post);
125 }
126
127 @Test // Slow
128 public void testRepeat3() {
129     final Timer timer = new Timer();
130     final int xxx = 20;
131     final double mean = timer.repeat(10, () -> xxx, t -> {
132         gotoSleep(t, 0);
133         return null;
134     }, t -> {
135         gotoSleep(t, -1);
136         return t;
137     }, t -> gotoSleep(10, 1));
138     assertEquals(10, new PrivateMethodTester(timer).invokePrivate("getLaps"));
139     assertEquals(xxx, mean, 30);
140     assertEquals(10, run);
141     assertEquals(10, pre);
142     assertEquals(10, post);
143 }
144
145 int pre = 0;
146 int run = 0;
147 int post = 0;
148
149 private void gotoSleep(long mSecs, int which) {

```

Failure Trace

terminated: TimerTest [Unit] C:\Program Files\Java\jdk-17.0.2\bin\java.exe (Feb 12, 2022, 11:24:32 PM - 11:24:35 PM)

```
2 * Copyright (c) 2017, Phasmid Software
4
5 package edu.neu.coe.info6205.sort.simple;
6
7 import edu.neu.coe.info6205.sort.*;
8
9 @SuppressWarnings("ALL")
10 public class InsertionSortTest {
11
12     @Test
13     public void sort0() throws Exception {
14         final List<Integer> list = new ArrayList<>();
15         list.add(1);
16         list.add(2);
17         list.add(3);
18         list.add(4);
19         Integer[] xs = list.toArray(new Integer[0]);
20         final Config config = ConfigTest.setupConfig("true", "0", "1", "", "");
21         Helper<Integer> helper = HelperFactory.create("InsertionSort", list.size(), config);
22         helper.init(list.size());
23         final PrivateMethodTester privateMethodTester = new PrivateMethodTester(helper);
24         final StatPack statPack = (StatPack) privateMethodTester.invokePrivate("getStatPack");
25         SortWithHelper<Integer> sorter = new InsertionSort<Integer>(helper);
26         sorter.preProcess(xs);
27         Integer[] ys = sorter.sort(xs);
28         assertTrue(helper.sorted(ys));
29         sorter.postProcess(ys);
30         final int compares = (int) statPack.getStatistics(InstrumentedHelper.COMPARES).mean();
31         assertEquals(list.size() - 1, compares);
32         final int inversions = (int) statPack.getStatistics(InstrumentedHelper.INVERSIONS).mean();
33         assertEquals(0L, inversions);
34         final int fixes = (int) statPack.getStatistics(InstrumentedHelper.FIXES).mean();
35         assertEquals(inversions, fixes);
36     }
37 }
```

Finished after 0.166 seconds

Runs: 4/4 Errors: 0 Failures: 0

edu.neu.coe.info6205.sort.simple.InsertionSortTest

- testMutatingInsertionSort (0.004 s)
- sort0 (0.012 s)
- sort1 (0.002 s)
- sort2 (0.006 s)

Failure Trace

Problems • Inavador Declaration Console

<terminated> InsertionSortTest [JUnit] C:\Program Files\Java\jdk-17.0.2\bin\java.exe (Feb 12, 2022, 11:25:57 PM - 11:25:58 PM)

2022-02-12 23:25:58 DEBUG Config - Config.get(helper, instrument) = true

2022-02-12 23:25:58 DEBUG Config - Config.get(helper, seed) = 0

2022-02-12 23:25:58 DEBUG Config - Config.get(instrumenting, copies) = true

2022-02-12 23:25:58 DEBUG Config - Config.get(instrumenting, swaps) = true

2022-02-12 23:25:58 DEBUG Config - Config.get(instrumenting, compares) = true

2022-02-12 23:25:58 DEBUG Config - Config.get(instrumenting, inversions) = 1

2022-02-12 23:25:58 DEBUG Config - Config.get(instrumenting, fixes) = true

2022-02-12 23:25:58 DEBUG Config - Config.get(helper, cutoff) =

Helper for InsertionSort with 4 elements

StatPack (copies: 0; inversions: 2,421; swaps: 2,421; fixes: 2,421; compares: 2,519)

5. Code:

```

public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> function, UnaryOperator<T> preFunction, Consumer<U> pc) {
    logger.trace("repeat: with " + n + " runs");
    // TO BE IMPLEMENTED: note that the timer is running when this method is called and should still be running when it returns
    // Pausing right when entering the repeat method
    pause();

    for(int i = 0; i < n; i++){
        // Calling the supplier method to generate an array (in case of insertion sort)
        // Supplier will be passed a method to generate different
        // sorts of arrays.
        T t = supplier.get();
        if(preFunction != null) {
            preFunction.apply(t);
        }

        // Resuming to time the insertion sort
        resume();

        // Sorting and storing output
        U u = function.apply(t);

        // Pausing the timer and lapping
        pauseAndLap();

        // Post Function
        if(postFunction != null) {
            postFunction.accept(u);
        }
    }

    // Calculating mean time and, resuming the timer and returning.
    double mean = meanLapTime();
    resume();
    return mean;
}
/**
 * Sort the sub-array xs:from:to using insertion sort.
 *
 * @param xs    sort the array xs from "from" to "to".
 * @param from  the index of the first element to sort
 * @param to    the index of the first element not to sort
 */
public void sort(X[] xs, int from, int to) {
    final Helper<X> helper = getHelper();

    // TO BE IMPLEMENTED

    if((to - from) < xs.length){
        System.out.println("Uneven length");
        return;
    }
    for (int i = from+1; i < to; i++)
    {
        int j=i;
        while(j > from && helper.less(xs[j],xs[j-1])){
            helper.swap(xs, j-1, j);
            j--;
        }
    }

    return;
}

```