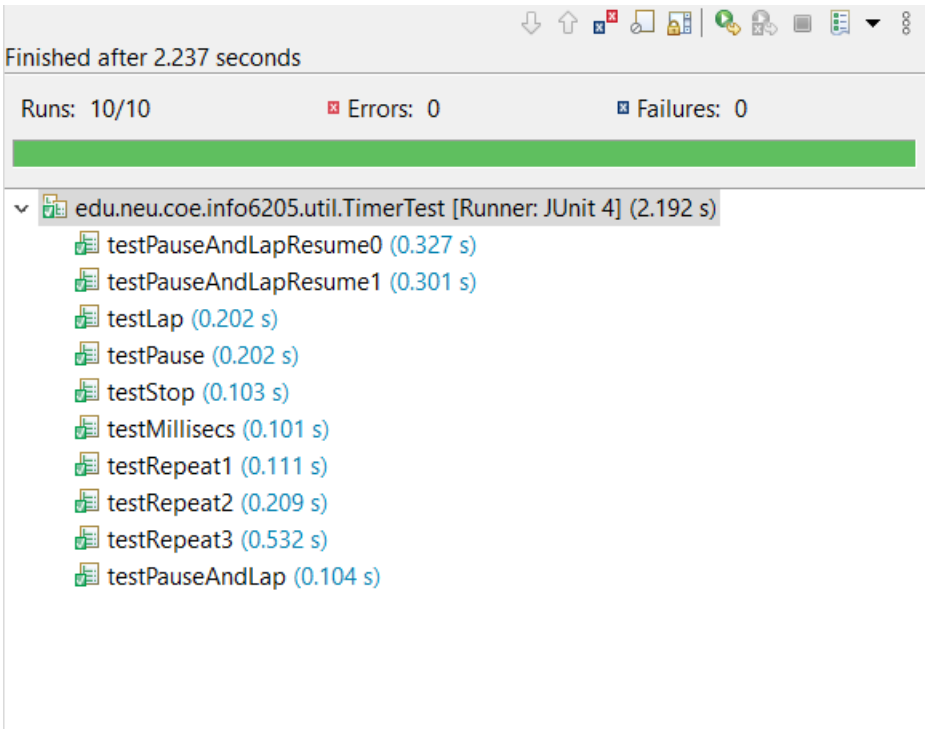
Task: To calculate the benchmark timing for InsertionSort function when introduced with arrays of different types. (Randomly ordered, ordered, partially ordered, reverse ordered) by implemeting the Timer class and InsertionSort function.

Part1: You are to implement three methods of a class called Timer. Please see the skeleton class in the repository. Timer is invoked from a class called Benchmark_Timerwhich implements the Benchmark interface.

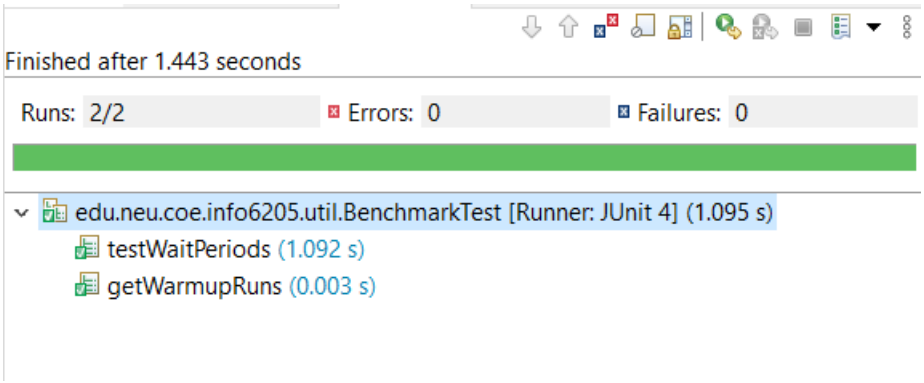
Timer.Java is loaded into the repository.

Testcases Execution:

TimerTest.java:



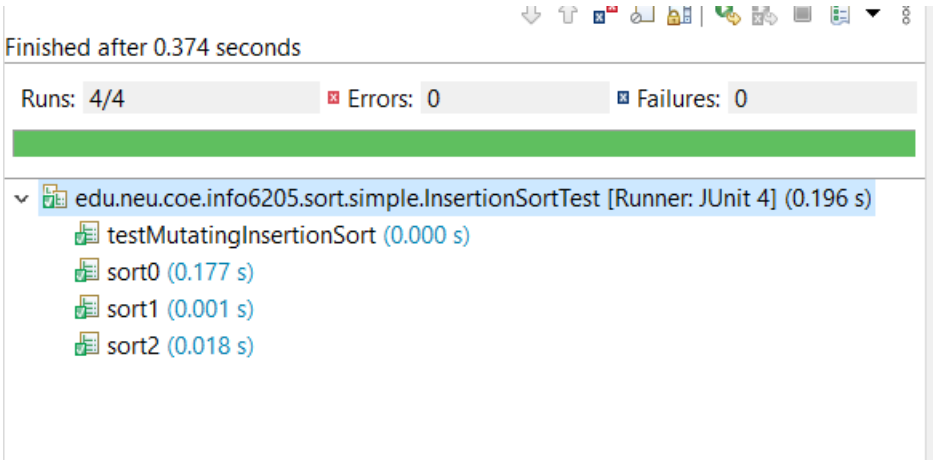
BenchmarkTest:



Part 2: Implement InsertionSort (in the InsertionSort class) by simply looking up the insertion code used by Arrays.sort. You should use the helper.swap method although you could also just copy that from the same source code. You should of course run the unit tests in InsertionSortTest.

InsertionSort.java loaded into the repository.

Testcases execution:



Part3: 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.

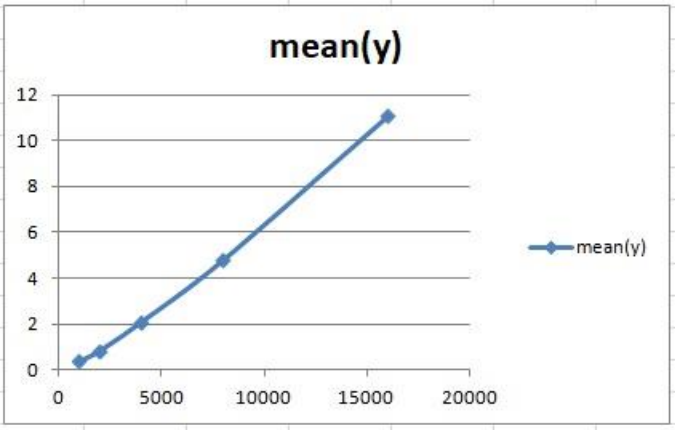
Main method implemented in Benchmark_Timer.java loaded into repository

```
    public static void main(String args[]) {

        InsertionSort is = new InsertionSort();
        Random random = new Random();
           for (int k = 1; k <= 4; k++) {
            int n = 1000;
            for (int i = 0; i < 5; i++) {
                Integer[] arr = new Integer[n];
                for (int j = 0; j < n; j++) {
                    arr[j] = random.nextInt();
                }
            switch(k)
            {
            case 2:
                Arrays.sort(arr);
                System.out.println("Mean for a sorted array of size " + n);
                break;
            case 4:
                 Arrays.sort(arr, Collections.reverseOrder());
                 System.out.println("Mean for a reverse ordered array of size " + n);
                 break;
            case 3:
                 Arrays.sort(arr, n/2, n);
                 System.out.println("Mean for partially sorted array of size " + n);
                 break;
            default :
                System.out.println("Mean for a randomly sorted array of size " + n);
                break;
            }
                // Arrays.sort(arr);

                int temp = n;
                Timer timer = new Timer();
                final double mean = timer.repeat(50, () -> 5, t -> {
                    is.sort(arr, 0, temp);
                    return null;
                });

                System.out.println(mean);

                n = n * 2;
            }
```

Findings for the programme:

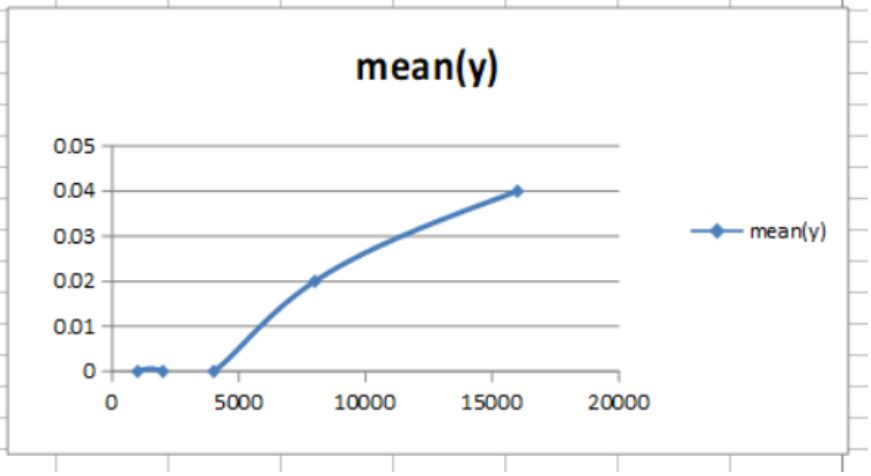
Case 1: (default) Random sorted array

arraysize(	mean(y)
1000	0.39
2000	0.86
4000	2.1
8000	4.82
16000	11.09



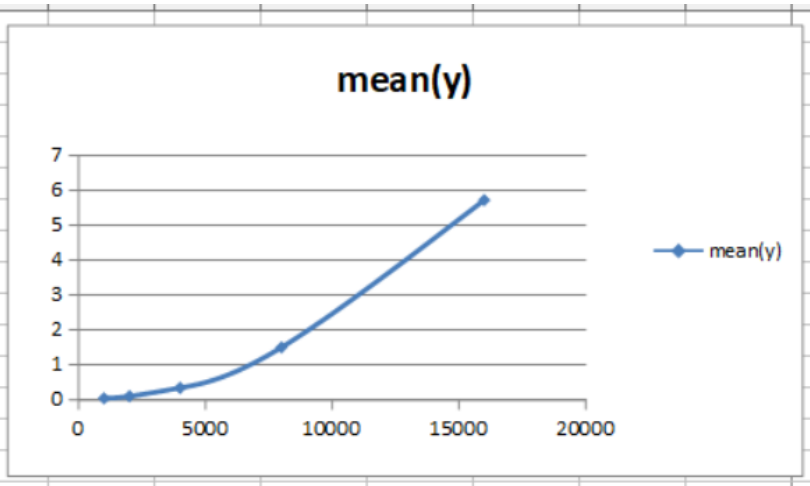
Case2: Sorted array

array size	mean(y)
1000	0
2000	0
4000	0
8000	0.02
16000	0.04



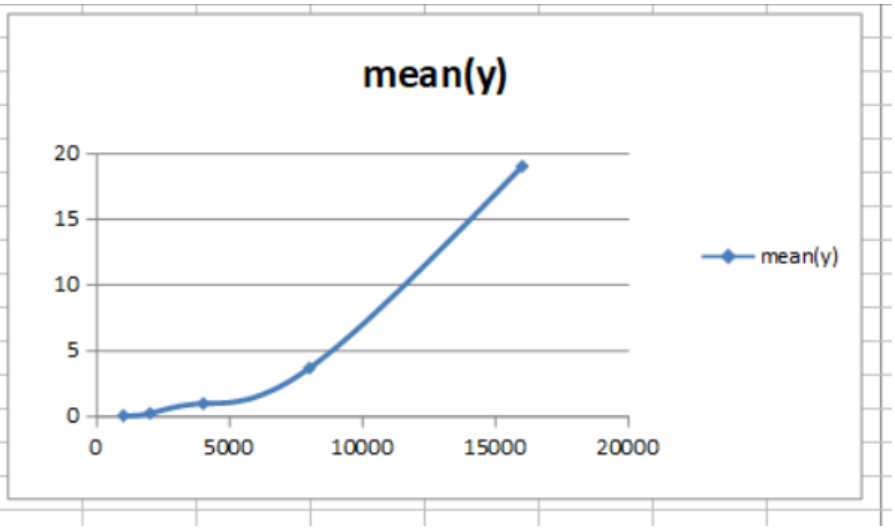
Case3: Partially sorted array:

array size(x)	mean(y)
1000	0.04
2000	0.1
4000	0.34
8000	1.5
16000	5.72

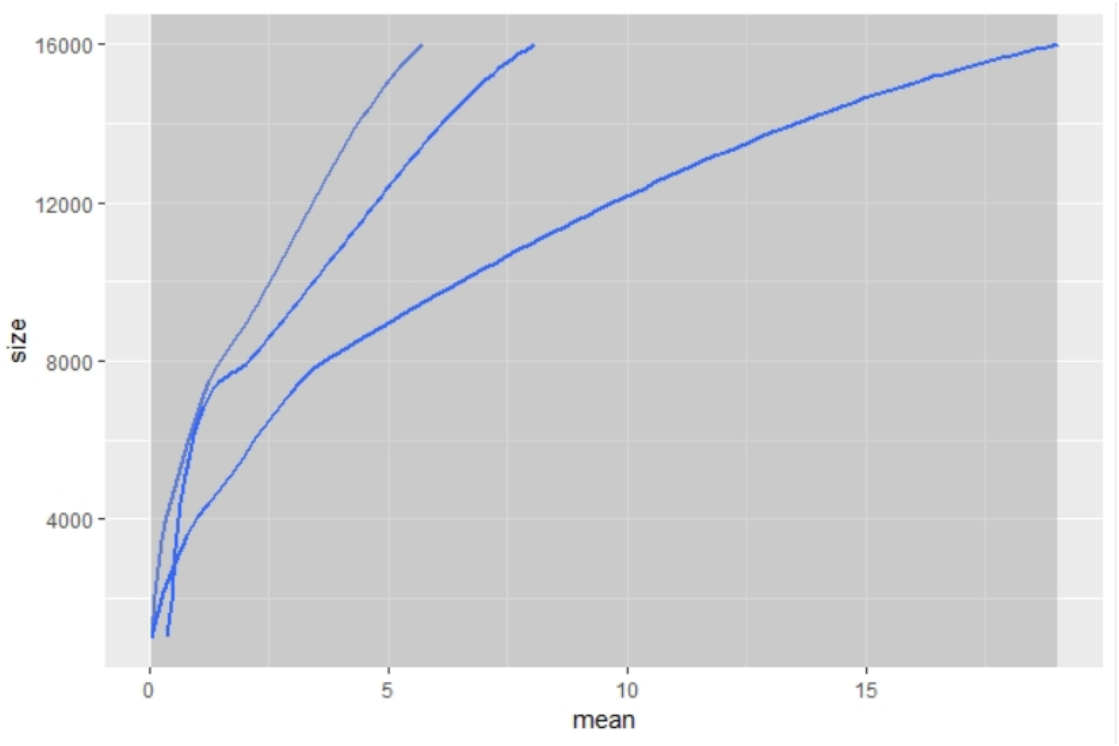


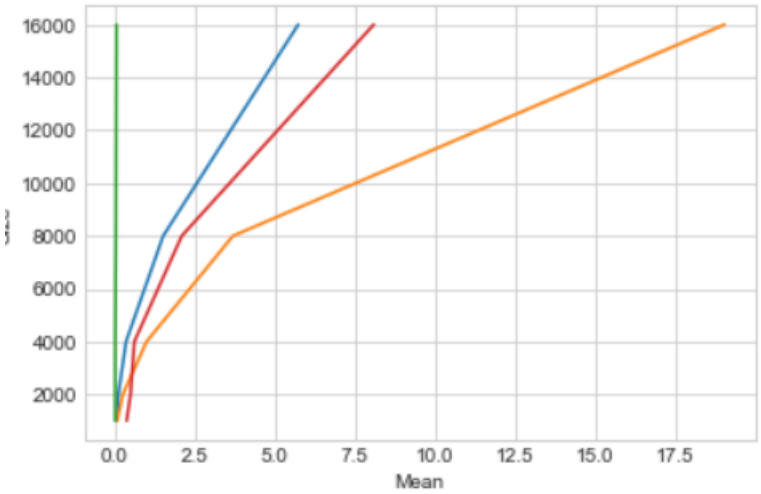
Case4: Reverse sorted array:

array size	mean(y)
1000	0.06
2000	0.24
4000	0.98
8000	3.68
16000	19.04



Comparision of time taken with respect to the mean time for all the four types of inputs given for InsertionSort function (R visualizaation) :





Legend:

Green - Sorted array as input which is faster than all.

Blue - Partially sorted

Red - Randomly Sorted

Orange - Reverse ordered which takes larger than all.

We can conclude that time taken by Insertion function for different inputs as follows:

Ordered < Partially Ordered < Randomly Ordered < Reverse Ordered.

Order of growth:

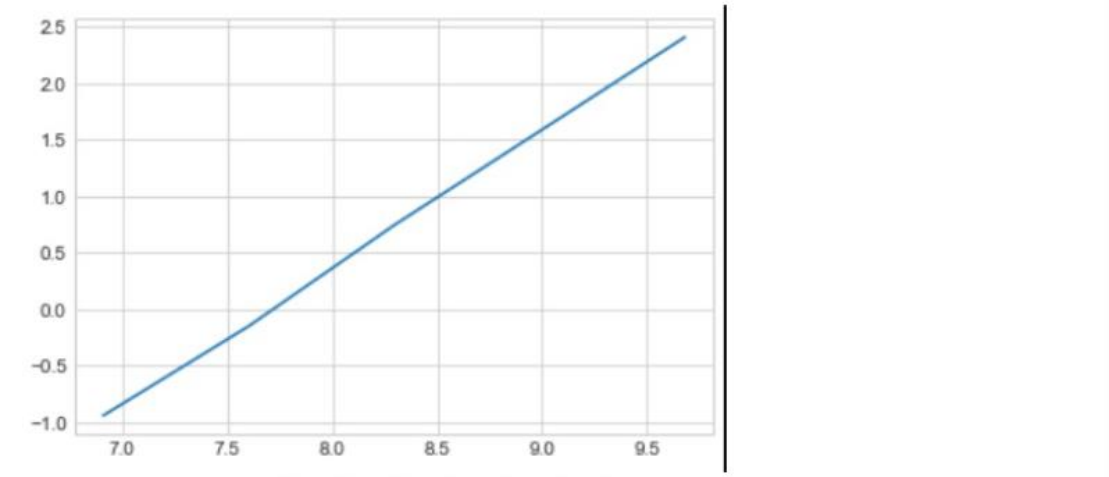
Let us consider the function of random sorted array:

Run the function for multiple times where u find values to be similarly linear

Put log - log graph on plot:

You will see a bit of linear function on graph:

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
import math
data = pd.read_csv('/Users/charan/Documents/PSA/Ass2/random.csv')
data
data["mean(y)"] = data["mean(y)"].apply(lambda x:math.log(x))
data["array size(x)"] = data["array size(x)"].apply(lambda x:math.log(x))
data
fig = plt.figure()
ax = plt.axes()

ax.plot(data['array size(x)'], data['mean(y)']);
```



By taking a sample elements of N1 = 1000 and t 1 = 0.39, and N2 = 2000 and t2 = 0.86 Order of growth can be t = N^k where k = 1.2.