PSA Assignment 3

Name- Padma Anaokar

Part A:

**Timer.java**

**public** <T, U> **double** repeat(**int** n, Supplier<T> supplier, Function<T, U> function, UnaryOperator<T> preFunction, Consumer<U> postFunction) {  
 ***logger***.trace(**"repeat: with "** + n + **" runs"**);  
 *//* ***FIXME: note that the timer is running when this method is called and should still be running when it returns. by replacing the following code  
 logger***.trace(**"repeat: with "** + n + **" runs"**);  
 T t = supplier.get();  
 pause();  
 **for** (**int** i = 0; i < n; i++) {  
 **if** (preFunction != **null**) {  
 t = preFunction.apply(t);  
 }  
 resume();  
 U u = function.apply(t);  
 pauseAndLap();  
 **if** (postFunction != **null**) {  
 postFunction.accept(u);  
 }  
  
 }  
 **double** meantime = meanLapTime();  
 resume();  
 **return** meantime;  
 *// END*}

**private static double** toMillisecs(**long** ticks) {  
 *//* ***FIXME by replacing the following code* double** milliSecs= (**double**)ticks/1000000;  
 **return** milliSecs;  
 *// END*}

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

**BenchMarkTest.java**

**Graphical user interface, text, application, email

Description automatically generated**

**TimerTest.java**

Graphical user interface, text, application, email

Description automatically generated

**Part B:**

Code changes in InsertionSort.java

**public void** sort(X[] xs, **int** from, **int** to) {  
 **final** Helper<X> helper = getHelper();  
 **for**(**int** i = from+1; i < to ; i++){  
 **int** k =i;  
 **while**(k > from && helper.swapStableConditional(xs, k) ){  
 k--;  
 }  
 }  
 *//* ***FIXME*** *// END*}

InsertionSortTest cases :

Graphical user interface, text, application

Description automatically generated

**Part C:**

**Time took to perform insertion sort on Different arrays**

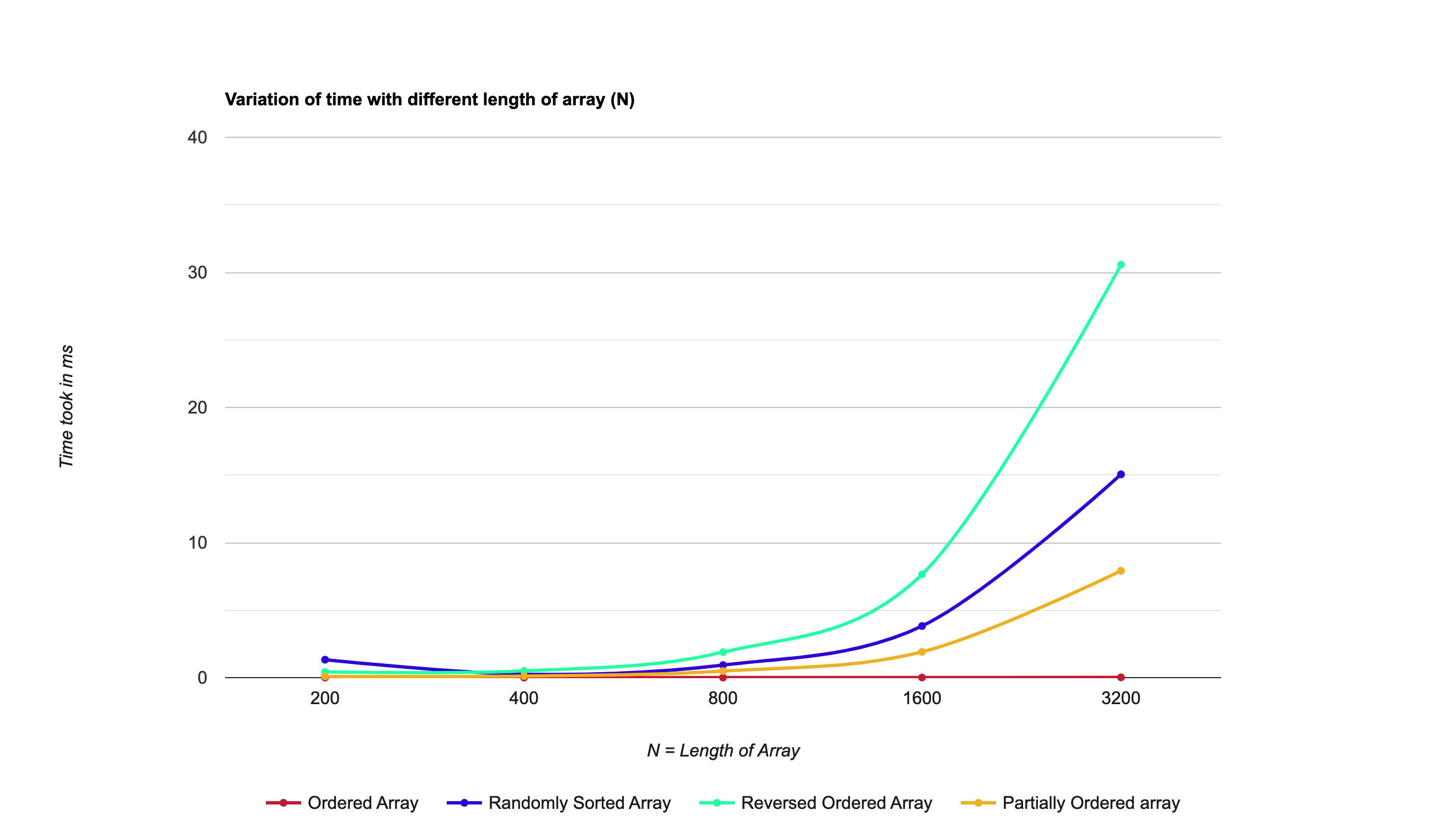
Code Changes in **InsertionSortBenchMark.java**

**public static void** main(String[] args) {  
  
 Random r = **new** Random();  
 InsertionSort insertionSort = **new** InsertionSort();  
  
 **for** (**int** n = 200; n <= 3200; n = n \* 2) {  
 *//Insertion sort on Randomly Ordered Array* ArrayList<Integer> randomArrayLst = **new** ArrayList<>();  
 **for** (**int** i = 0; i < n; i++) {  
 randomArrayLst.add(r.nextInt(n));  
 }  
 Integer[] randomIntArray = randomArrayLst.toArray(**new** Integer[0]);  
 Benchmark<Boolean> benchmarkRandomArr = **new** Benchmark\_Timer<>(  
 **"Running Insertion sort on :Randomly Ordered Array "**, b -> {  
 insertionSort.sort(randomIntArray.clone(), 0, randomIntArray.**length**);  
 });  
 **double** randomArrayResult = benchmarkRandomArr.run(**true**, 10);  
  
 *//Insertion sort on Ordered Array* ArrayList<Integer> orderedArrayList = **new** ArrayList<>();  
 **for** (**int** i = 0; i < n; i++) {  
 orderedArrayList.add(i);  
 }  
 Integer[] orderedArr = orderedArrayList.toArray(**new** Integer[0]);  
 Benchmark<Boolean> benchmarkOrderedArr = **new** Benchmark\_Timer<>(  
 **"Running Insertion sort on :Ordered Array "**, b -> {  
 insertionSort.sort(orderedArr.clone(), 0, orderedArr.**length**);  
 });  
 **double** orderedArrayResult = benchmarkOrderedArr.run(**true**, 10);  
  
 *//Insertion sort on Reverse Ordered Array* ArrayList<Integer> reverseArrayList = **new** ArrayList<>();  
 **for** (**int** i = 0; i < n; i++) {  
 reverseArrayList.add(n - i);*//reverseList[100-00]= reverseList[100],reverseList[99]* }  
 Integer[] reverseIntArray = reverseArrayList.toArray(**new** Integer[0]);  
 Benchmark<Boolean> benchmarkReverseArr = **new** Benchmark\_Timer<>(  
 **"Running Insertion sort on :Reversely Ordered Array"**, b -> {  
 insertionSort.sort(reverseIntArray.clone(), 0, reverseIntArray.**length**);  
 });  
 **double** reversedArrayResult = benchmarkReverseArr.run(**true**, 10);  
  
 *//Insertion sort on Partially Ordered Array* ArrayList<Integer> partialArrayList = **new** ArrayList<>();  
 **for** (**int** j = 0; j < n; j++) {  
 **if** (j > n / 2) {  
 partialArrayList.add(r.nextInt(n));*//randomly arranged* } **else** {  
 partialArrayList.add(j);*//ordered array* }  
 }  
 Integer[] partialIntArray = partialArrayList.toArray(**new** Integer[0]);  
 Benchmark<Boolean> benchmarkPartial = **new** Benchmark\_Timer<>(  
 **"Running Insertion sort on :Partially Ordered Array"**, b -> {  
 insertionSort.sort(partialIntArray.clone(), 0, partialIntArray.**length**);  
 });  
 **double** partialArrayResult = benchmarkPartial.run(**true**, 10);  
  
 System.***out***.println(**"N is : "** + n);  
 System.***out***.println(**"Random Array takes : "** + randomArrayResult+**"ms"**);  
 System.***out***.println(**"Ordered Array takes : "** + orderedArrayResult +**"ms"**);  
 System.***out***.println(**"Reversed Array takes : "** + reversedArrayResult+**"ms"**);  
 System.***out***.println(**"Partial Array takes : "** + partialArrayResult+**"ms"**);  
 }

**Graphs:**

**Time Taken for different length and differently Ordered Array**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **Random** | **Ordered** | **Partially-Ordered** | **Reverse** |
| 200 | 1.33 | 0.005 | 0.092 | 0.43 |
| 400 | 0.25 | 0.003 | 0.118 | 0.5 |
| 800 | 0.93 | 0.006 | 0.488 | 1.89 |
| 1600 | 3.82 | 0.011 | 1.908 | 7.64 |
| 3200 | 15.05 | 0.022 | 7.913 | 30.57 |

****

**Separate Graphs:**

Ordered Array:

A picture containing line chart

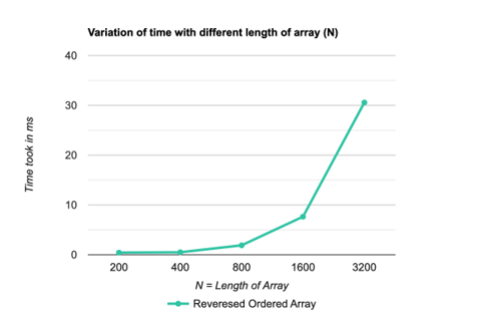
Description automatically generated

Randomly Ordered Array:

Chart, line chart

Description automatically generated

Reversed ordered Array:



Partially Ordered Array:

Chart, line chart

Description automatically generated

* Graphs attached in Excel Sheet as well

**Conclusion:**

* It can be observed from the graph that the reverse ordered array takes most time for larger array sizes when sorting it using insertion sort because it must move all the elements to its respective places
* Random array takes a little lesser time than the reverse ordered array since there are high chances that the random numbers generated the random function have some of the numbers already in the correct position
* Partially sorted order takes more lesser time than the random array since half of the array would already be in the sorted position.
* The best case would be an already sorted array which takes the least time which is expected as well since the array is already in the sorted order.