Chandrakanth Chittappa (001300076)  
Program Structures & Algorithms  
Fall 2021  
Assignment No. 2

Task

Your task for this assignment is in three parts.

* (Part 1) You are to implement three methods 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);  
}

public class Benchmark\_Timer<T> implements Benchmark<T> {

public Benchmark\_Timer(String description, UnaryOperator<T> fPre, Consumer<T> fRun, Consumer<T> fPost)

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  
}

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.

Don't forget to check your implementation by running the unit tests in *BenchmarkTest* and *TimerTest*. If you have trouble with the exact timings in the unit tests, it's quite OK (in this assignment only) to change parameters until the tests run. Different machine architectures will result in different behavior.

* (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.

As usual, the submission will be your entire project (*clean, i.e. without the target and project folders).*There are stubs and unit tests in the repository.

Report on your observations and show screenshots of the runs and also the unit tests. Please note that you may have to adjust the required execution time for the insertion sort unit test(s) because your computer may not run at the same speed as mine.

Further notes: you should use the *System.nanoTime* method to get the clock time. This isn't guaranteed to be accurate which is one of the reasons you should run the experiment several times for each value of *n*. Also, for each invocation of *run*, run the given target function ten times to get the system "warmed up" before you start the timing properly.

The *Sort* interface takes care of copying the array when the *sort(array)* signature is called. It returns a new array as a result. The original array is unchanged. Therefore, you do not need to worry about the insertion-based sorts getting quicker because of the arrays getting more sorted (they don't).

Part1

Screen shot of TimerTest cases passing

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Screen shot of BenchmarkTest cases passing



Part 2

Relationship Conclusion

Insertion Sort Time complexity

Since insertion sort loops throughout the elements in the left of a particular element to insert the current element in the proper position (2 for loops). The worst case of Insertion sort is O (N 2) where N is length of the array.

Screen Shots and console output

Graphical user interface, text, application, email

Description automatically generated

Screen shot of Insertion Test cases passing

Output from the console while running InsertionSortTest

2021-09-24 19:42:04 DEBUG Config - Config.get(helper, instrument) = true

2021-09-24 19:42:04 DEBUG Config - Config.get(helper, seed) = 0

2021-09-24 19:42:04 DEBUG Config - Config.get(instrumenting, copies) = true

2021-09-24 19:42:04 DEBUG Config - Config.get(instrumenting, swaps) = true

2021-09-24 19:42:04 DEBUG Config - Config.get(instrumenting, compares) = true

2021-09-24 19:42:04 DEBUG Config - Config.get(instrumenting, inversions) = 1

2021-09-24 19:42:04 DEBUG Config - Config.get(instrumenting, fixes) = true

2021-09-24 19:42:04 DEBUG Config - Config.get(instrumenting, hits) = true

2021-09-24 19:42:04 DEBUG Config - Config.get(helper, cutoff) =

Helper for InsertionSort with 4 elements

StatPack {hits: 9,684; copies: 0; inversions: 2,421; swaps: 2,421; fixes: 2,421; compares: 2,519}

StatPack {hits: 19,800; copies: 0; inversions: 4,950; swaps: 4,950; fixes: 4,950; compares: 4,950}

Part 3

**Output in the console while running the main method in Benchmark\_Timer class**Text

Description automatically generated with low confidence

Time taken to run



**Drawing a graph from the above values:**

Chart

Description automatically generated

As we can see the order of the arrays are follows:

1. Ordered array [This array sorts very quickly, much faster than the worst-case complexity for larger values of n]
2. Random array [This array is slower than the ordered array but faster than the rest]
3. Partially random array [This array is faster than the reversed ordered array but slower than the rest]
4. Reversed Sorted array [This array is the slowest among the arrays whose curve is much closer to the N^2 curve which is the worst time complexity of insertion sort]