**Programing Structures and Algorithms**

**Spring 2023 (Sec-01)   
Assignment 3**

**Name :- Swarag Sanjay Gutte**

**NUID:- 002728422**

**Problem statement:-**

Your task for this assignment is in three parts.

* (Part 1) You are to implement three (3) methods (*repeat*, *getClock*, and *toMillisecs*) 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);  
}

[*Supplier* is a Java function type which supplies values of type *T* using the method: *get()*.]

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

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

[*Consumer<T>* is a Java function type which consumes a type *T* with the method: *accept*(t).

*UnaryOperator<T>* is essentially an alias of *Function<T, T>* which defines *apply(t)* which takes a *T* and returns a *T*.]

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  
}

[*Function<T, U>* defines a method *U apply(t: T)*, which takes a value of *T* and returns a value of *U*.]

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.

**OUTPUT**  
Unit Tests  
**1. Insertion Sort**

Text

Description automatically generated

**2. Timer**

Text

Description automatically generated

**3. Benchmark**

Text

Description automatically generated

**Benchmark Results:**

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

**Relationship Conclusion:**

Insertion sort is a stable and adaptive sorting algorithm, meaning it maintains the relative order of equal elements and performs well on partially sorted arrays. The number of comparisons and swaps required by insertion sort grows as N(N-1)/2 on average, where N is the number of elements in the array. A sorted array will take the least amount of time to sort, while a reverse-sorted array will take the longest due to the number of swaps required. A partially sorted or randomly sorted array will have a lower number of swaps, making the sorting process faster.

Hence It can be stated that

Ordered Array < Partially Ordered Array < Randomly Ordered Array < Reverse Ordered Array.

**Evidence** : -

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | Reverse Array | Random Array | Parallel Array | Order Array |
| 20 | 1.02 | 1.02 | 0.25 | 0.14 |
| 40 | 0.28 | 0.28 | 0.13 | 0.3 |
| 80 | 0.16 | 0.16 | 0.15 | 0.15 |
| 160 | 0.17 | 0.17 | 0.13 | 0.11 |
| 320 | 0.29 | 0.29 | 0.23 | 0.15 |
| 640 | 0.82 | 0.82 | 0.87 | 0.16 |
| 1280 | 3.1 | 3.1 | 2.16 | 0.09 |
| 2560 | 10.78 | 9.78 | 8.11 | 0.09 |
| 5120 | 42.74 | 37.74 | 32.12 | 0.1 |
| 10240 | 171.26 | 141.26 | 130.43 | 0.35 |
| 20480 | 680.02 | 620.02 | 521.87 | 0.16 |
| 40960 | 2736.49 | 2378.41 | 2072.26 | 0.24 |

