Yuan Lian (001063563)

**Program Structures & Algorithms**

**Fall 2021**

**Assignment No. 2**

* **Task**
* (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.
* **Relationship Conclusion:**

When N increase, the running time of sort goes up as well.In most cases the running time of “reversed” > “random” > “partially” > “ordered”. But it could happen differently like when N = 16000.

* **Evidence to support the conclusion:**

The output of running time is the evidence.

* 1. **Output**

Part3 Result

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | N=1000 | N=2000 | N=4000 | N=8000 | N=16000 |
| Reversed | 0.888041375 | 0.856574325 | 0.6635352 | 0.890817775 | 0.5572184 |
| Random | 0.7788863 | 0.4890436 | 0.336565 | 0.780025325 | 0.456470275 |
| Partially | 0.467245675 | 0.380935125 | 0.322844525 | 0.588065925 | 0.50453775 |
| Ordered | 0.351332425 | 0.320704125 | 0.28224865 | 0.52472615 | 0.408465475 |

**2.Graphical Representation**

* **Unit tests result:**

BenchmarkTest

图形用户界面, 文本

描述已自动生成

TimerTest

文本

描述已自动生成

InsertionSortTest

图形用户界面, 文本

描述已自动生成