COS 126	Princeton University	Fall 2024

Programming Exam

Before you begin. Read through this page of instructions. Do *not* start the exam (or access the next page) until instructed to do so.

Duration. You have 80 minutes to complete this exam.

Advice. *Review the entire exam before starting to write code. Implement the constructor and methods in the order given, one at a time, testing in* `main()` *after you complete each method.*

Submission. Submit your solutions on *TigerFile* using the link from the *Exams* page. You may submit multiple times (but only the last version will be graded).

Check Submitted Files. You may click the *Check Submitted Files* button to receive *partial* feedback on your submission (up to 10 times). We will attempt to provide this feature during the exam (but you should not rely upon it).

Grading. Your program will be graded on correctness, efficiency, and clarity. You will receive partial credit for a program that implements some of the required functionality. *You will receive a substantial penalty for a program that does not compile.*

Allowed resources. This exam is *open-book* but *not* open-internet. During the exam you may use only the following resources: course textbook; companion booksite; lecture slides; course website; your course notes; your code from the programming assignments or precept; course *Ed*; course *codePost*, *Java visualizer*, and *Oracle Javadoc*. Accessing other websites or resources is prohibited. For example, you may not use *Google*, *Google Docs*, *StackOverflow*, or *ChatGPT*.

No collaboration or communication. During the exam, collaboration and communication (including sharing files) are prohibited, except with course staff members. A staff member will be outside the exam room to answer clarification question.

No electronic devices or software. Software and computational/communication devices are prohibited, except to the extent needed for taking this exam (such as a laptop, browser, and IntelliJ). For example, you must close all unnecessary virtual desktops, applications, and browser tabs; disable notifications; and *power off* all other devices (such as cell phones, tablets, smart watches, and earbuds). You *must* use only the Princeton wireless network *eduroam*, not a mobile hotspot or other network.

Honor Code pledge. Write and sign the Honor Code pledge by typing the text below in the file `acknowledgments.txt`.

> *I pledge my honor that I will not violate the Honor Code during this examination.*

Electronically sign it by typing `/s/` followed by your name.

After the exam. Discussing or communicating the contents of this exam before solutions have been posted is a violation of the Honor Code, as is accessing *TigerFile*.

Deliverables. For this programming exam, you will submit three files:

1. A Java program `LapTimer.java`, containing a data type for recording lap times in a race.
2. A Java program `LapTimerClient.java`, containing a client program.
3. A text file `acknowledgments.txt`, containing your Honor Code pledge.

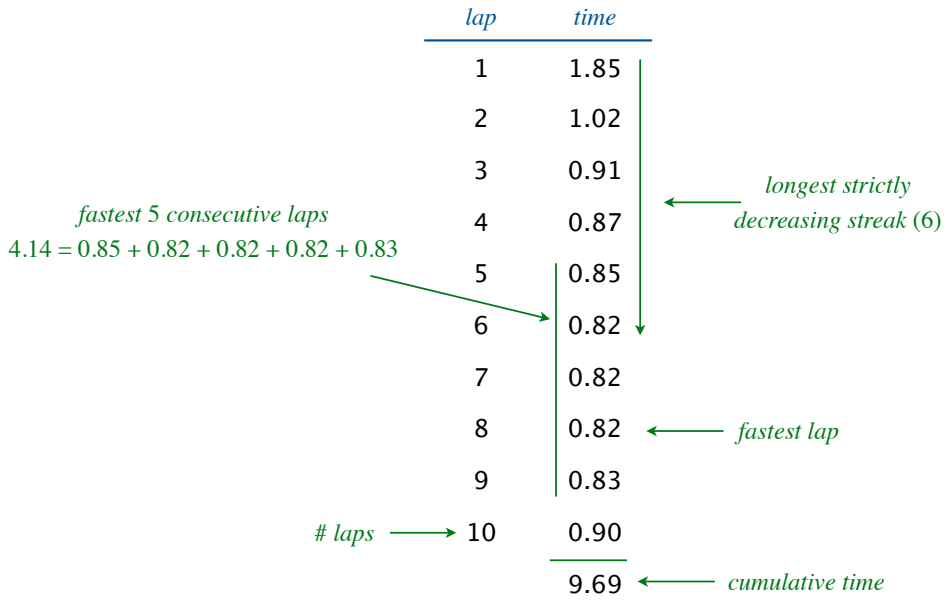
Lap timer. The data type `LapTimer` models a device that records *lap times* in a race. The race is divided into *laps*, each of the same length. For example, a swimmer might swim 1,500 meters by completing 30 laps in a 50-meter pool; a distance runner might run 10,000 meters by completing 25 laps on a 400-meter track. The device records the time to complete each successive lap.

API. Using the template file `LapTimer.java` provided in the project folder as a starting point, write a data type that implements the following API:

`public class LapTimer`

`public LapTimer(int max)`	*creates a new timer that supports up to* `max` *laps*
`public void addLap(double time)`	*adds a new lap with the specified time*
`public int count()`	*number of laps added*
`public double cumulativeTime()`	*cumulative time of all laps*
`public String toString()`	*string representation of this timer (format specified below)*
`public double fastestLap()`	*fastest time of any lap*
`public double fastestMultiLap(int k)`	*fastest cumulative time of any k consecutive laps*
`public int longestDecreasingStreak()`	*maximum length of a sequence of consecutive laps in which the lap times strictly decrease*
`public static void main(String[] args)`	*unit tests the* `LapTimer` *data type (code provided)*

Example. For example, the diagram below shows the times for each 10-meter segment (or "lap") in a race by Usain Bolt, in which he ran 100 meters in 9.69 seconds. The diagram also identifies a number of relevant statistics.



API details. Here is some additional information about the required behavior:

- The `toString()` method returns a string that represents the lap timer, with the (unformatted) lap times delimited by commas and spaces, and enclosed in square braces. Here is an example:

  ```
  [1.85, 1.02, 0.91, 0.87, 0.85, 0.82, 0.82, 0.82, 0.83, 0.9]
  ```

- The `fastestMultiLap()` method returns the total time it took to complete the fastest k *consecutive* laps. It could arise from the first k laps, the last k laps, or any other group of k consecutive laps.

- The `longestDecreasingStreak()` method returns the *maximum length* of any sequence of *consecutive* laps for which each lap time is *strictly less* than the previous one. It is an integer between 1 and `count()`.

- *Exceptions.* Throw an `IllegalArgumentException` if the argument to `addLap()` is invalid:

  – Calls `addLap()` with a `time` argument that is less than (or equal to) zero.

  – Calls `addLap()` if `max` laps have already been added.

  On this exam, for simplicity, you may assume that a client program always:

  – Calls the constructor with a `max` argument that is a positive integer.

  – Calls `addLap()` at least once before calling any other instance method.

  – Calls `fastestMultiLap()` with an argument `k` that is between 1 and `count()`.

- *Unit testing.* Include a `main()` method that directly calls the constructor and every instance method, such as the following:

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

    // create a lap timer for Usain Bolt's 100 meter world record,
    // dividing the race into 10 segments (or "laps")
    LapTimer timer = new LapTimer(10);
    timer.addLap(1.85);
    timer.addLap(1.02);
    timer.addLap(0.91);
    timer.addLap(0.87);
    timer.addLap(0.85);
    timer.addLap(0.82);
    timer.addLap(0.82);
    timer.addLap(0.82);
    timer.addLap(0.83);
    timer.addLap(0.90);

    // print statistics
    StdOut.println("number of laps  = " + timer.count());                    // 10
    StdOut.println("cumulative time = " + timer.cumulativeTime());           // 9.69
    StdOut.println("fastest lap     = " + timer.fastestLap());               // 0.82
    StdOut.println("fastest 50m     = " + timer.fastestMultiLap(5));         // 4.14
    StdOut.println("longest streak  = " + timer.longestDecreasingStreak());  // 6

    // print times: [1.85, 1.02, 0.91, 0.87, 0.85, 0.82, 0.82, 0.82, 0.83, 0.9]
    StdOut.println(timer);
}
```

- *Performance requirements.* For full credit, the instance methods must meet (or exceed) the following worst-case running time requirements, where n is the number of laps added:

instance method	*time*
`addLap()`	$\Theta(1)$
`count()`	$\Theta(1)$
`cumulativeTime()`	$\Theta(1)$
`toString()`	$\Theta(n^2)$

instance method	*time*
`fastestLap()`	$\Theta(n)$
`fastestMultiLap()`	$\Theta(n)$
`longestDecreasingStreak()`	$\Theta(n)$

- *Floating-point precision.* You need not worry about floating-point precision issues that might arise in either `cumulativeTime()` or `fastestMultiLap()`.

Restrictions. You may use classes defined only in `java.lang` (such as `Math` and `Double`) or in our textbook input/output libraries (such as `StdOut` and `In`). So, for example, you may *not* use classes defined in `java.util` (such as `java.util.ArrayList` and `java.util.Arrays`).

Client program. Write a client program `LapTimerClient.java` that takes the name of a file as a command-line argument, creates a `LapTimer` object, adds the lap times in the file to the `LapTimer` object, and prints the length of a longest decreasing streak.

The input file format consists of the number of laps n, followed by the n lap times, with each value on its own line. Here are some sample executions:

```
~/Desktop/f24-pe>  more bolt100m.txt
10
1.85
1.02
0.91
0.87
0.85
0.82
0.82
0.82
0.83
0.90

~/Desktop/f24-pe> java-introcs LapTimerClient bolt100m.txt
6

~/Desktop/f24-pe> java-introcs LapTimerClient agnel400m.txt
3

~/Desktop/f24-pe> java-introcs LapTimerClient ledecky1500m.txt
6
```

Grading. This programming exam is worth a total of 50 points. Here is the breakdown:

part	*points*
`LapTimer()`	6
`addLap()`	6
`count()`	3
`cumulativeTime()`	4
`toString()`	5

part	*points*
`fastestLap()`	4
`fastestMultiLap()`	8
`longestDecreasingStreak()`	8
`LapTimerClient`	6

You may earn full credit for `LapTimerClient.java` even if `LapTimer.java` is not fully functional.