CS2030 Randomized Arrival and Service Time (Question)

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Discrete Event Simulator Version 3 — Randomized Arrival and Service Time

Requirements

- OOP Design: Inheritance-based event dispatch
- Use `PriorityQueue` from the Java Collections Framework to store the events;
- Packaging the classes into the package `cs2030.simulator`;
- Randomizing the arrival and service times.

Priority Queuing

The `PriorityQueue` class can be used to keep a collection of elements, where each element is given a certain priority.

- Elements can be added with the `add(E e)` method;
- The `poll()` method can be used to retrieve and remove the element with the highest priority; it returns an object of type `E`, or `null` if the queue is empty.

To enable `PriorityQueue` to order events, instantiate a `PriorityQueue` object using the constructor that takes in a *Comparator* object. *For more details, refer to the Java API Specifications.*

Using Packages

Recall that Java has a higher-level of abstraction barrier called a package. So far, you have been creating classes in the default package. In this lab, you are to create the `cs2030.simulator` package. A directory `cs2030/simulator` has been created for you in your labs, it contains a `RandomGenerator` class that will be used for the

next part.

Specifically, all classes dealing with the simulation should now reside in the `cs2030.simulator` package, with the `Main` class importing the necessary classes from the package.

Randomized Arrival and Service Time

Simulation of different arrival and service times is achieved via the use of random number generation. A random number generator is an entity that generates one random number after another. Since it is not possible to generate a truly random number algorithmically, pseudo random number generation is adopted instead. A pseudo-random number generator can be initialized with a seed, such that the same seed always produces the same sequence of (seemingly random) numbers.

Although, Java provides a class `java.util.Random`, an alternative `RandomGenerator` class that is more suitable for discrete event simulation is provided for you that encapsulates different random number generators for use in our simulator. Each random number generator generates a different stream of random numbers. The constructor for `RandomGenerator` takes in three parameters:

- `int seed` is the base seed. Each random number generator uses its own seed that is derived from this base seed;
- `double lambda` is the arrival rate, λ;
- `double mu` is the service rate, μ.

The **inter-arrival time** is usually modeled as an exponential random variable, characterized by a single parameter λ denoting the arrival rate. The `genInterArrivalTime()` method of the class `RandomGenerator` is used for this purpose. Specifically,

- start the simulation by generating the first arrival event with timestamp `0`.
- every time an arrival event is processed, it generates another arrival event and schedules it;
- if there are still more customers to simulate, generate the next arrival event with a timestamp of `T + now`, where `T` is generated with the method `genInterArrivalTime()`;

The **service time** is modeled as an exponential random variable, characterized by a single parameter, service rate μ. The method `genServiceTime()` from the class `RandomGenerator` can be used to generate the service time. Specifically,

- each time a customer is being served, a `DONE` event is generated and scheduled;
- the `DONE` event generated will have a timestamp of `T + now`, where `T` is generated with the method `genServiceTime()`.

You may refer to the API of the `RandomGenerator` class here. The class file can be downloaded from here.

As usual, you will need to keep track of the following statistics:

1. the average waiting time for customers who have been served
2. the number of customers served
3. the number of customers who left without being served

The Task

Input now consists of the following (in order of presentation):

- the first line is an `int` value denoting the base seed for the `RandomGenerator` object;
- the second line is an `int` value representing the number of servers
- the third line is an `int` representing the number of customers (or the number of arrival events) to simulate
- the fourth line is a `double` parameter for the arrival rate, λ
- the last line is a `double` parameter for the service rate, μ

Output comprises the individual discrete events, and also the statistics at the end of the simulation.

Take note of the following assumptions:

- The maximum number of customers is 1000;
- The format of the input is always correct;
- Output of a `double` value, say `d`, is to be formatted with `String.format("%.3f", d)`;

As this lab is a continuation form the previous one, your programs for `sim1` to `sim6` has been made available to you. Just remember to

- define a `Main` class with the `main` method to handle input;
- check for output format correctness using the `diff` utility (see specific level for usage details). Note that only **one** test case is provided for this;
- check for styling errors by invoking `checkstyle`. For example, to check styling for all java files

```
$ checkstyle *.java
```

- save a copy of all source files into the appropriate level directory (see specific level for usage details).

Level 1

Implementing the `PriorityQueue`

This level aims to provide you with a sanity check of your correctness in using the `PriorityQueue`. You will need to instantiate a `PriorityQueue` object using the constructor that takes in a `Comparator` object.

Create your own `EventComparator` class and implement the `compare` method.

```
class EventComparator implements Comparator {
    public int compare(Event e1, Event e2) {
       :
    }
}
```

You should run your existing program first, then re-implement the queue with a `PriorityQueue` and check for consistency in output.

The following is a sample run of the program. User input is underlined.

```
1
0.500
0.600
0.700
1.500
1.600
1.700
0.500 1 arrives
0.500 1 served by 1
0.600 2 arrives
0.600 2 waits to be served by 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 served by 1
1.500 4 arrives
1.500 4 waits to be served by 1
```

```
1.600 5 arrives
1.600 5 leaves
1.700 6 arrives
1.700 6 leaves
2.500 2 done serving by 1
2.500 4 served by 1
3.500 4 done serving by 1
[0.633 3 3]
```

```
2
0.500
0.600
0.700
1.500
1.600
1.700
0.500 1 arrives
0.500 1 served by 1
0.600 2 arrives
0.600 2 served by 2
0.700 3 arrives
0.700 3 waits to be served by 1
1.500 1 done serving by 1
1.500 3 served by 1
1.500 4 arrives
1.500 4 waits to be served by 1
1.600 2 done serving by 2
1.600 5 arrives
1.600 5 served by 2
1.700 6 arrives
1.700 6 waits to be served by 2
2.500 3 done serving by 1
2.500 4 served by 1
2.600 5 done serving by 2
2.600 6 served by 2
3.500 4 done serving by 1
3.600 6 done serving by 2
[0.450 6 0]
```

Check the format correctness of the output by typing the following Unix command

```
$ java Main < test12.in | diff - test1.out
```

Make a copy of your Java programs to the level directory by typing the Unix commands

```
$ mkdir rng1
$ cp *.java rng1
```

Level 2

Creating the `cs2030.simulator` package

This level is another sanity check of your correctness during package creation.

To place all classes dealing with the simulation into the `cs2030.simulator` package, add the following line into the classes (except the `Main` class).

```
package cs2030.simulator;
```

You should also change the appropriate access modifiers to reflect in-package assess.

The `Main` class should now import the necessary classes in the package. Note that you should only import the specific classes that the `Main` class depends on. Do not use `import cs2030.simulator.*;`

Compile the program with the following:

```
javac -d . *.java
```

and you will find that the compiled classes will be deposited into `cs2030/simulator` automatically.

The following is a sample run of the program. User input is underlined.

```
1
0.500
0.600
0.700
1.500
1.600
1.700
0.500 1 arrives
0.500 1 served by 1
0.600 2 arrives
0.600 2 waits to be served by 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 served by 1
1.500 4 arrives
1.500 4 waits to be served by 1
1.600 5 arrives
1.600 5 leaves
1.700 6 arrives
1.700 6 leaves
2.500 2 done serving by 1
2.500 4 served by 1
3.500 4 done serving by 1
[0.633 3 3]
```

```
2
0.500
0.600
0.700
```

```
1.500
1.600
1.700
0.500 1 arrives
0.500 1 served by 1
0.600 2 arrives
0.600 2 served by 2
0.700 3 arrives
0.700 3 waits to be served by 1
1.500 1 done serving by 1
1.500 3 served by 1
1.500 4 arrives
1.500 4 waits to be served by 1
1.600 2 done serving by 2
1.600 5 arrives
1.600 5 served by 2
1.700 6 arrives
1.700 6 waits to be served by 2
2.500 3 done serving by 1
2.500 4 served by 1
2.600 5 done serving by 2
2.600 6 served by 2
3.500 4 done serving by 1
3.600 6 done serving by 2
[0.450 6 0]
```

Check the format correctness of the output by typing the following Unix command

```
$ java Main < test12.in | diff - test2.out
```

Make a copy of your Java programs to the level directory by typing the Unix commands

```
$ mkdir rng2
$ cp *.java rng2
```

Level 3

Randomizing inter-arrival time

In this level, we shall randoming the inter-arrival time while remaining with the constant service time of `1.0`.

Note that the `RandomGenerator.class` has been created for the `cs2030.simulator` package and placed in the `cs2030/simulator` directory.

The program takes as input a seed value, the number of servers, the number of customers, arrival rate and service rate. The program then outputs the individual discrete events, and the statistics at the end of the simulation.

The following is a sample run of the program. User input is underlined.

```
1
1
5
1.0
1.0
0.000 1 arrives
0.000 1 served by 1
0.314 2 arrives
0.314 2 waits to be served by 1
1.000 1 done serving by 1
1.000 2 served by 1
1.205 3 arrives
1.205 3 waits to be served by 1
2.000 2 done serving by 1
2.000 3 served by 1
2.776 4 arrives
2.776 4 waits to be served by 1
3.000 3 done serving by 1
3.000 4 served by 1
3.877 5 arrives
3.877 5 waits to be served by 1
4.000 4 done serving by 1
4.000 5 served by 1
5.000 5 done serving by 1
[0.366 5 0]
```

```
1
2
10
1.0
1.0
0.000 1 arrives
0.000 1 served by 1
0.314 2 arrives
0.314 2 served by 2
1.000 1 done serving by 1
1.205 3 arrives
1.205 3 served by 1
1.314 2 done serving by 2
2.205 3 done serving by 1
2.776 4 arrives
2.776 4 served by 1
3.776 4 done serving by 1
3.877 5 arrives
3.877 5 served by 1
3.910 6 arrives
3.910 6 served by 2
4.877 5 done serving by 1
4.910 6 done serving by 2
9.006 7 arrives
9.006 7 served by 1
9.043 8 arrives
9.043 8 served by 2
```

```
9.105 9 arrives
9.105 9 waits to be served by 1
9.160 10 arrives
9.160 10 waits to be served by 2
10.006 7 done serving by 1
10.006 9 served by 1
10.043 8 done serving by 2
10.043 10 served by 2
11.006 9 done serving by 1
11.043 10 done serving by 2
[0.178 10 0]
```

Check the format correctness of the output by typing the following Unix command

```
$ java Main < test34.in | diff - test3.out
```

Make a copy of your Java programs to the level directory by typing the Unix commands

```
$ mkdir rng3
$ cp *.java rng3
```

Level 4

Include randomizing service time

Finally, add in the randomization of service time.

The program takes as input a seed value, the number of servers, the number of customers, arrival rate and service rate. The program then outputs the individual discrete events, and the statistics at the end of the simulation.

The following is a sample run of the program. User input is underlined.

```
1
1
5
1.0
1.0
0.000 1 arrives
0.000 1 served by 1
0.313 1 done serving by 1
0.314 2 arrives
0.314 2 served by 1
0.417 2 done serving by 1
1.205 3 arrives
1.205 3 served by 1
1.904 3 done serving by 1
2.776 4 arrives
2.776 4 served by 1
```

```
2.791 4 done serving by 1
3.877 5 arrives
3.877 5 served by 1
4.031 5 done serving by 1
[0.000 5 0]
```

```
1
2
10
1.0
1.0
0.000 1 arrives
0.000 1 served by 1
0.313 1 done serving by 1
0.314 2 arrives
0.314 2 served by 1
0.417 2 done serving by 1
1.205 3 arrives
1.205 3 served by 1
1.904 3 done serving by 1
2.776 4 arrives
2.776 4 served by 1
2.791 4 done serving by 1
3.877 5 arrives
3.877 5 served by 1
3.910 6 arrives
3.910 6 served by 2
3.922 6 done serving by 2
4.031 5 done serving by 1
9.006 7 arrives
9.006 7 served by 1
9.043 8 arrives
9.043 8 served by 2
9.105 9 arrives
9.105 9 waits to be served by 1
9.160 10 arrives
9.160 10 waits to be served by 2
10.484 7 done serving by 1
10.484 9 served by 1
10.781 9 done serving by 1
11.636 8 done serving by 2
11.636 10 served by 2
11.688 10 done serving by 2
[0.386 10 0]
```

Check the format correctness of the output by typing the following Unix command

```
$ java Main < test34.in | diff - test4.out
```

Make a copy of your Java programs to the level directory by typing the Unix commands

```
$ mkdir rng4
$ cp *.java rng4
```

Submission (Course)

Select course: CS2030 (2018/2019 Sem 2) - Programming Methodology II ▼

Your Files: BROWSE

SUBMIT (only .java, .c, .cpp, .h, and .py extensions allowed)

To submit multiple files, click on the Browse button, then select one or more files. The selected file(s) will be added to the upload queue. You can repeat this step to add more files. Check that you have all the files needed for your submission. Then click on the Submit button to upload your submission.