ITI 1121. Introduction to Computing II
Winter 2024

Assignment 2

Mehrdad Sabetzadeh and Tommaso Cesari
(Last modified on February 3, 2024)

Deadline for Assignment 2: February 26, 2024, 11:30 pm

Introduction

In this assignment, you will implement a simple parking-lot simulator. The parking lot, depicted in Figure 1, has one entrance and one exit. The entrance gate can let cars in only one at a time (i.e., no simultaneous entrances). Similarly, the exit gate can let out only one at a time (i.e., no simultaneous exits). The parking lot is managed via two queues: one for the cars wanting to enter (incoming queue), and one for the cars wanting to exit (outgoing queue).

The simulation is driven by a simulated clock. In this assignment, the clock ticks every second. **Note that one second on the simulated clock does *not* correspond to an actual second on the wall clock.** The simulated clock is simply a counter that gets incremented by one unit in each iteration of the simulation. In each simulation run, we will simulate a total duration of 24 hours; that corresponds to $3600 * 24 = 86400$ simulation iterations (i.e., 86400 simulated seconds). On a laptop, the entire simulation will take approximately $\frac{1}{2}$ second of wall-clock time.

To assist you with the implementation, this assignment is broken down into four tasks, numbered 1 to 4, below. Please keep in mind that the tasks are *not* of equal sizes. Tasks 1-3 are conceptually easier. Task 4 is slightly more involved and comparatively requires more effort. We encourage you to start early! For this assignment, you can reuse the code you wrote in Assignment 1 (A1) as well as the reference A1 solution that we provide.

Figure 1: Illustration of a parking lot with one entrance and one exit

Task 1. Complete the `TrangularDistribution` Class.

The implementation of the `TrangularDistribution` class is a good place to start because it is entirely independent of the next tasks. A triangular distribution is a probability distribution with a lower limit a, an upper limit b and a mode c, where $a < c < b$ [1]. We illustrate in Figure 2 the shape of the probability density function (pdf) for a triangular distribution.

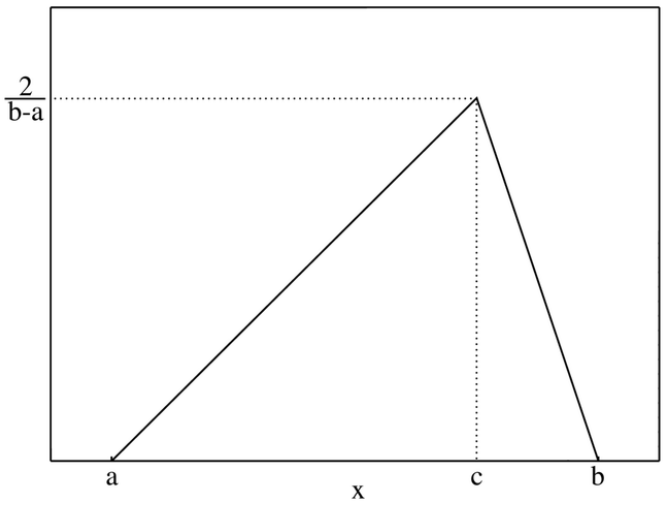

Figure 2: Probability density function for a triangular distribution (source: Wikipedia)

For any given value x, the probability density function, $pdf(x)$, is computed as follows:

$$pdf(x) = \begin{cases} 0 & \text{for } x < a, \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \leq x < c, \\ \frac{2}{b-a} & \text{for } x = c, \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c < x \leq b, \\ 0 & \text{for } b < x. \end{cases}$$

In Task 1, you will implement the constructor and the *pdf* function in the `TriangularDistribution` class.

```
public TriangularDistribution(int a, int c, int b) {
 // Your code goes here ...

}

public Rational pdf(int x) {

 // Your code goes here ...

}
```

For the purposes of this assignment, we want to keep the resulting probability values as rational numbers. That is why the `pdf(...)` method returns an instance of `Rational`. In Lab 3, you will have already seen and worked with a simple implementation of `Rational`. A more complete implementation of this class is provided to you in the starter code. In this assignment, you do *not* need to make any changes to the `Rational` class.

[1] The distribution simplifies when $c = a$ or $c = b$, but we are not concerned with these special cases. In this assignment, we deal only with the situation where $a < c < b$.

To illustrate, the concept of a triangular distribution, suppose that $a = 0$, $c = 5$, and $b = 10$. We get: $pdf(\leq 0) = 0$, $pdf(1) = \frac{1}{25}$, $pdf(2) = \frac{2}{25}$, $pdf(3) = \frac{3}{25}$, $pdf(4) = \frac{4}{25}$, $pdf(5) = \frac{1}{5}$, $pdf(6) = \frac{4}{25}$, $pdf(7) = \frac{3}{25}$, $pdf(8) = \frac{2}{25}$, $pdf(9) = \frac{1}{25}$, and $pdf(\geq 10) = 0$.

An important property of our distribution is that $\sum_{x=a}^{x=b} pdf(x) = 1$. In other words, the sum of the probabilities add up to 1, just as we would expect. The `TrangularDistribution` class has a `main(...)` method with some additional examples for testing. The output from `TrangularDistribution.main(...)` is provided to you alongside the starter code to help you test your implementation.

Now, let us see how a triangular distribution is going to help us with our parking-lot simulation: **we use a triangular distribution to capture the probability that a car will leave the lot after being parked for a certain duration**. In our simulation, we set the maximum parking duration to 8 hours; you can assume that any car exceeding 8 hours of parking will be immediately towed out! We further set the mode of our distribution to 4 hours, i.e., $c = 4 * 3600$, noting that the our clock ticks in seconds. If you look at the variable declarations in the starter code of the `Simulator` class, you will see the following:

```
private static final int MAX_PARKING_DURATION = 8 * 3600;
.
.
.
private static final TriangularDistribution departurePDF =
new TriangularDistribution(0, MAX_PARKING_DURATION / 2, MAX_PARKING_DURATION);
```

In Task 4, you will use `departurePDF`, defined above, to calculate in each iteration of the simulation how probable it is for each parked car to want to leave the parking lot. This probability is *not* calculated based on what simulation iteration we are in, but rather based on the duration that any given car has been parked in the lot.

The fact that we now need to keep track of the duration of parking for each car necessitates some changes to our A1 design of the parking lot. More specifically, we need to somehow store a "timestamp" alongside each car parked in the lot to know when the car entered the lot and got parked. This is what we are going to do in Task 2 by defining a `Spot` class.

Task 2. Complete the `Spot` class.

This task is simple: you need to complete the constructor and the getter / setter methods of the `Spot` class. An instance of `Spot` stores not only a reference variable of type `Car` but also a primitive integer, `timestamp`, to keep track of when a car got parked in the lot. Through this timestamp, we can calculate the duration that a car has been parked in the lot and subsequently how probable the car is to leave at any point in time. We further use the `Spot` class to capture arrivals in the entrance and exit queues. When instances of `Spot` are put into either of the two queues, the `timestamp` value denotes the time (simulation iteration) when the associated car was put in the queue.

Task 3. Complete / revise the `ParkingLot` class.

For this task, you will mostly reuse what you did in A1 yourself or alternatively build on the A1 reference solution provided to you. There are a number of important differences between `ParkingLot` in A1 and `ParkingLot` in A2:

- Occupancy is no longer tracked using a two-dimensional array of cars. Instead, a two-dimensional array of `Spots` is used, so that we can record when each car was parked in the lot. This change necessitates a number of minor enhancements to the methods that you implemented previously in `ParkingLot`.
- The input file to be processed by the constructor of `ParkingLot` no longer contains any occupancy information. In A2, the input file contains only the parking-lot design information. We will see in Tasks 4 where the cars that want to park in the lot are coming from. **In A2, you will no longer read the occupancy information from a file.**
- You will implement a new method `public boolean attemptParking(Car c, int timestamp)`. This method, which will be used for the implementation of Task 4, checks whether there is *any* suitable spot in the lot for car `c` based on `c`'s type and the car-type restrictions that need to be observed (e.g., large cars being allowed to park only in large spots). The car-type restrictions are exactly as defined in A1. If a suitable spot is found, the following happens: (1) `c` is parked in the lot and the associated timestamp is set to the `timestamp` parameter of `attemptParking(...)`; (2) the method call will return `true`.

```
1.  Set the simulation clock to zero; // this clock counts in seconds (s)

2.  Repeat until the simulation clock has reached 24h (i.e., 24 * 3600s) {
3.    Probabilistically determine whether a car arrives this very second (yes / no);
4.    If a car arrives this second, generate a random car and place it in the incoming queue;

5.    For every car c parked in the lot {
6.      Calculate the duration that c has been parked;

7.      If (duration == 8h) remove the car from the lot and place it in the outgoing queue;
8.      Else (i.e., when duration < 8h) {
9.        Probabilistically determine whether c leaves this very second (yes / no);
10.       If c is leaving, remove the car from the lot and put in the outgoing queue;
11.     }
12.   }

13.   If the incoming queue is not empty, attempt to park the first car from the this queue;

14.   If the outgoing queue is not empty, let the first car in this queue out;

15.   Increase the simulation clock by 1s;
16. }
```

Figure 3: The simulation algorithm (pseudo-code)

Otherwise, if no suitable spot is available for c, the call to `attemptParking(...)` returns `false`, meaning that the parking lot does not currently have a suitable spot to accommodate c. This in turn means that c has to wait until a suitable spot becomes available, i.e., until a currently parked car leaves and the vacated spot is able to accommodate c based on its type. For example, if c is of type LARGE, it has to wait until a LARGE spot becomes vacant before c is allowed into the lot.

IMPORTANT!: For completing Task 3, please start with the starter code provided for A2 and bring what you need from the A1 implementation of `ParkingLot` one method at a time. Do *not* start with your A1 implementation as your basis. Some variable declarations and method signatures have changed since A1; you would not want to get stuck due to such changes going potentially unnoticed.

Task 4. Complete the `Simulator` class.

Task 4 is concerned with implementing the simulation algorithm. The pseudo-code for the algorithm you need to implement is provided in Figure 3.

Important Considerations for Task 4.

The algorithm is straightforward and self-explanatory. There are nonetheless some important technical considerations that you need to take note of to successfully implement each part. These considerations, listed below, are organized by the line numbers in the algorithm of Figure 3:

- **Line 3**: To calculate the probability that a car arrives at any given second, you need to note the following: when the `Simulator` class is instantiated, a per-hour arrival rate (`perHourArrivalRate`) is specified. For simplicity, we assume that incoming car arrivals are distributed evenly over time. The probability that a car arrives at any given second is therefore `perHourArrivalRate/3600`. To probabilistically determine whether a car has arrived at a given second, you can make use of the `RandomGenerator.eventOccurred(...)` method provided to you. Remember to set `probabilityOfArrivalPerSec` (instance variable) in `Simulator`'s constructor. Once that has been done, you can call `RandomGenerator.eventOccurred(probabilityOfArrivalPerSec)` to probabilistically determine whether or not a car arrives at a given second.

- **Line 4**: To generate a random car, you can use the `RandomGenerator.generateRandomCar()` method. This method has hard-coded proportions for different car types: ELECTRIC (5%), SMALL (20%), LARGE (10%), and REGULAR (default). We assume these proportions to be fixed for this assignment.
- **Lines 4, 7, 10**: The queues in this assignment expect instances of `Spot`. Every time an instance of `Spot` is enqueued, the `timestamp` variable of the instance is set to the current simulation time (i.e., `clock`).
- **Line 9**: To probabilistically determine whether a parked car is going to leave the lot at a given second, you can use `RandomGenerator.eventOccurred(...)`. The parameter you pass to this method is however *different* from what was explained for Line 3. On Line 9, you need to calculate the probability based on how long the car in question has been parked. As stated earlier, we assume that the duration of parking follows a triangular distribution. **You therefore have to calculate the probability of departure by calling `departurePDF.pdf(duration)`**, where `duration` is the amount of (simulated) time that a given car has been parked in the lot.
- **Line 10**: Note that the outgoing queue has a rather marginal role in this assignment: All we are achieving through the outgoing queue is to ensure that if multiple cars decide to leave at exactly the same time (i.e., the same clock tick), we are not going to let them leave all at once. Instead, exiting cars have to queue up and be let out one at a time.
- **Line 13, consideration 1**: Before attempting to park the car that is at the front (head) of the incoming queue, we need to dequeue an element from the incoming queue. This is because our `Queue` interface does not allow us to "peek into" the queue without dequeuing first. As a result, until an element has been dequeued, we cannot know the type of the car that is requesting to be parked and thus cannot check whether the parking lot has a suitable spot for that car. If parking is attempted (through the `attemptParking(...)` method of `ParkingLot`), but the attempt fails (i.e., `attemptParking(...)` returns `false`), the dequeued element cannot be put back in the queue; otherwise, the element will go to the back of the queue and thus lose its priority! Instead, we need to retain the most recently dequeued element until a suitable spot becomes available. As long as this element has not been parked, no other element is dequeued from the incoming queue. In other words, no cars will be rejected from parking and no car loses its priority in the queue. Parking will be attempted repeatedly every (simulated) second until the dequeued car can be parked.
- **Line 13, consideration 2**: If the attempted parking is successful, a record of entry must be printed to the standard output. Let `c` be the car that has been successfully parked. The record of entry should be produced by the following statement:

```
System.out.println(c + " ENTERED at timestep " + clock + "; occupancy is at " +
                   lot.getTotalOccupancy()
                  );
```

- **Line 14**: When an element is dequeued from the outgoing queue, a record of exit must be printed to the standard output. Let `c` be the car that has been dispatched from the exit queue. The record of exit should be produced by the following statement:

```
System.out.println(c + " EXITED at timestep " + clock + "; occupancy is at " +
                   lot.getTotalOccupancy()
                  );
```

Implementation

We are now ready to program our solution. Just like in A1, you need to follow the patterns provided to you in the template code. To ensure you follow the rules on what is allowed and what is not, please consult post @34 on Piazza. You can complete Tasks 1-4 above in any order you like. However, we encourage you to follow the order in which the tasks appear (that is, Task 1, then Task 2, and so on). Like in A1, guidance is provided through comments in the template code. The locations where you need to write your code are clearly indicated with an inline comment which reads as follows:

```
// WRITE YOUR CODE HERE!
```

Running A2

The `main(...)` method of the `Simulator` has been provided to you in its entirety in the starter code. You should use the following command line to run A2:

```
java Simulator <lot-design filename> <hourly rate of arrival>

Example usage: java Simulator parking.inf 11
```

Please feel free to alter the `main(...)` method as you debug and test different parts of your implementation. **Nevertheless, when you submit A2, please make sure to include the original `main(...)` method provided in the starter code.**

Example Output

Example outputs are provided in a directory named `outputs` alongside the starter code.

Academic Integrity

This part of the assignment is meant to raise awareness concerning plagiarism and academic integrity. Please read the following documents.

- https://www.uottawa.ca/about/14-fraude-scolaire
- https://www.uottawa.ca/vice-president-academic/academic-integrity

Cases of plagiarism will be dealt with according to the university regulations. By submitting this assignment, you acknowledge:

1. I have read the academic regulations regarding academic fraud.
2. I understand the consequences of plagiarism.
3. With the exception of the source code provided by the instructors for this course, all the source code is mine.
4. I did not collaborate with any other person, with the exception of my partner in the case of team work.
    - If you did collaborate with others or obtained source code from the Web, then please list the names of your collaborators or the source of the information, as well as the nature of the collaboration. Put this information in the submitted README.txt file. Marks will be deducted proportional to the level of help provided (from 0 to 100%).

Rules and regulation

- Follow all the directives available on the assignment directives web page.
- Submit your assignment through the on-line submission system virtual campus.
- You must preferably do the assignment in teams of two, but you can also do the assignment individually.
- You must use the provided template classes below.
- We will be using an automated tool to compare all the assignments against each other (this includes both, the French and English sections). Submissions that are flagged by this tool will receive the grade of 0.
- It is your responsibility to make sure that Brightspace has received your assignment. Late submissions will not be graded.

Files

You must hand in a **zip** file (**no other file format will be accepted**). The name of the top directory has to have the following form: **a2_3000000_3000001**, where 3000000 and 3000001 are the student numbers of the team members submitting the assignment (simply repeat the same number if your team has one member). The name of the folder starts with the letter "a" (lowercase), followed by the number of the assignment, here 2. The parts are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. The archive a2_3000000_3000001.zip (available on Brightspace under Assignment 2) contains the files that you can use as a starting point. Your submission must contain the following files.

- README.txt
  - A text file that contains the names of the two partners for the assignments, their student ids, section, and a short description of the assignment (one or two lines).
- Car.java (submit as-is without changes)
- CarType.java (submit as-is without changes)
- LinkedQueue.java (you are **not** allowed to change this class; submit as-is without changes; we will implement LinkedQueue together later on in the term)
- **ParkingLot.java**
- Queue.java (you are **not** allowed to change this interface; submit as-is without changes; we will discuss this interface when we start talking about ADTs)
- RandomGenerator.java (submit as-is without changes)
- Rational.java (submit as-is without changes)
- **Simulator.java**
- **Spot.java**
- **StudentInfo.java** (Update the `display()` method)
- **TriangularDistribution.java**
- Util.java (submit as-is without changes)

IMPORTANT! If you are working in a team of two, please **make sure that only *one* team member makes a submission**. This means that you need to coordinate in advance with your teammate who is going to submit. If both members of the same team make a submission, then both members will be penalized for double submission.

Questions

For all your questions, please visit the Piazza Web site for this course:

- https://piazza.com/uottawa.ca/winter2024/iti1121/

Last modified: February 3, 2024