ITI 1121. Introduction to Computing II
Winter 2024

Assignment 3

Mehrdad Sabetzadeh and Tom Cesari
(Last modified on February 28, 2024)

Deadline for Assignment 3: March 22, 2024, 11:30 pm

Introduction

In this assignment, we continue with the parking-simulator theme established in Assignments 1 and 2. Our goal in Assignment 3 (A3) is as follows: knowing a priori the car arrival rate and how long cars stay in the lot, we would like to determine the minimum number of spots that the parking lot should have to avoid excessive queue buildup at the entrance. What we do in A3 represents a real-world application for simulation: when a real parking lot is being designed, an informed decision needs to be made about how many spots are needed. If there are too few spots, the parking lot will be overwhelmed; in our simulation, such a situation manifests itself as an unacceptably large queue buildup at the parking entrance. Just as having too few spots is problematic, having too many is problematic too! If the capacity of the parking lot exceeds demand by a large margin, the lot will be under-utilized. If this happens, the operational costs of the lot may exceed the revenue that the lot generates, potentially making the lot financially unviable. Finding a balance between supply and demand is therefore essential.

What we aim to calculate in A3 is "just the right number of spots", given an hourly car arrival rate and the probability distribution for how long a car stays parked once it has entered the lot. For simplicity, like in A2, we assume that the probability distribution for departures is fixed. The code that you develop in A3 therefore has only one user-defined parameter: the hourly car arrival rate. Provided with this parameter, your implementation should simulate the parking lot for different numbers of spots and determine the lowest number of spots that meets demand (we discuss what we mean by "meeting demand" a bit later).

Like in A2, a simulation run spans 24 (simulated) hours. However, in contrast to A2, we need to run the simulation *numerous* times rather than just once. Specifically, we need to progressively try larger numbers of spots $1, 2, \ldots$, all the way to the number of spots that would meet demand. For example, suppose we set the hourly rate to 3 cars per hour. To determine how large our lot should be for this level of demand, we start by simulating a lot that has only one spot, then a lot that has only two spots, then a lot that has only three spots, and so on. Let us assume that, through this process, we determine that our lot needs 15 spots to meet demand. This means that we will have simulated the parking lot 15 times, each time for 24 (simulated) hours.

The question that now arises is whether we can trust a single simulation run? For example, if we run the simulation only once with 15 spots and somehow we see that there is no queue buildup at the entrance, can we have reasonable confidence that 15 spots would be sufficient? The answer is No! Why? Because our simulation is probabilistic. Like in every probabilistic process, we need to mitigate *random variation*, that is, the chances of being too lucky or too unlucky in a single run. And, how do we mitigate random variation? By repeating the simulation process several times. A common strategy is to have 10 repetitions and consider the average results. Going back to our example in the previous paragraph, to account for random variation, we need to run the simulation 10 times for a lot that has one spot, 10 times for a lot that has two spots, and so on. This means that we will have simulated the parking lot 150 times (as opposed to merely 15 times) in the scenario described earlier where we found 15 spots would meet demand.

In A3, what we want to obtain from an individual simulation run is the number of cars left in the entrance queue after 24 (simulated) hours. To accept a number n of spots as meeting demand, we repeat for 10 times the simulation with n spots. If the average queue size (across 10 runs) is below a given threshold, we can be reasonably confident that having n spots is sufficient. For this assignment, **we define "meeting demand" as yielding an average entrance queue length of ≤ 5 across 10 simulation runs**. In other words, if we repeat the simulation with n spots for 10

```
1.    Initialize lot size (n) to 1;

2.     Repeat {

3.       Do 10 times {
4.         Simulate the lot for 24 hours at the (user-provided) car arrival rate;
5.       }

6.       Take the average of incoming queue sizes across the 10 simulation runs;

7.       if (average <= 5) {

           // The lot is large enough to meet demand
8.         break out of the Repeat loop;

9.       } else {

          // The lot is still not large enough to meet demand
10.       n = n + 1;

11.     }
12.   }
```

Figure 1: Algorithm (pseudo-code) for determining the number n of spots to meet parking demand

times, take the average of the queue length at the entrance (after 24 simulated hours), and this average is ≤ 5, then we are done and return n as meeting demand.

Figure 1 provides the pseudo-code for finding the *smallest* number of spots n that meets demand. You get to implement this algorithm in Task 4, below. To simplify the implementation and further to give you more exposure to the new concepts we will see in class (particularly lists), various tweaks need to be made to the code you previously wrote in A2 before you can adapt that code for A3. To assist you with the implementation, this assignment is broken down into five tasks, numbered 1 to 5, as discussed below. Please keep in mind that the tasks are *not* of equal sizes. For A3, you can reuse the code you wrote in A2 as well as the reference A2 solution provided to you.

Task 1. Complete the `ParkingLot` class.

The `ParkingLot` class is going to be simplified considerably in A3. First, there is no lot design anymore: we assume that any car can park at any spot. The `CarType` enumeration is therefore gone from A3. We further let go of the rectangular (2D-array) design of the lot. The `ParkingLot` class now has a `capacity` instance variable (primitive integer) storing the *maximum number of cars* that the lot can accommodate. Occupancy is now stored in an instance variable, `occupancy`, which is of type *List*. We use the `LinkedList` implementation of `List` in A3.

In Task 1, you get to implement selected methods of `ParkingLot` using lists (as opposed to arrays). These methods that you need to implement (or enhance) are:

- `public ParkingLot(int capacity)`
- `public void park(Car c, int timestamp)`
- `public Spot remove(int i)`
- `public boolean attemptParking(Car c, int timestamp)`
- `public Spot getSpotAt(int i)`
- `public int getOccupancy()`

For details about what the methods in our new version of `ParkingLot` should do, please consult the documentation and comments in the A3 starter code.

Task 2. Implement `size()` and `peek()` in the `LinkedQueue` class.

In A2, not being able to simply get the size of a queue posed a challenge. There, we had to dequeue all the elements to count how many elements were in the queue. In A3, we have enhanced the `Queue` interface to provide a `size()` method. We have also enhanced `Queue` with a `peek()` method to allow us to peek at the element at the front of the queue. Recall that in A2, when we dequeued a car and the attempt at parking that car failed, we needed special treatment for that now-dequeued element. We can provide a cleaner implementation of our simulator if we have a `peek()` method for queues, similar to what we saw in class for stacks. In Task 2, you get to implement `size()` and `peek()` for the linkedlist-based implementation of queues, `LinkedQueue`, which we will have seen in class. To keep track of the queue size, you will need to define an additional instance variable.

Task 3. Update the `Simulator` class.

Modify the `simulate()` method in the `Simulator` class to work with the revised interface of `ParkingLot` and the new `peek()` method in `Queue`. In addition, implement the `getIncomingQueueSize()` method of `Simulator` using the `size()` method of `Queue`. The `getIncomingQueueSize()` is going to be used in the `CapacityOptimizer` class (next task) to determine the size of the incoming queue after a simulation run.

IMPORTANT!: To complete Tasks 1 and 3, please start with the starter code provided for A3 and bring what you need from A2 one method at a time. Do *not* start with your A2 implementation. Several small changes have been made since A2; you would not want to get stuck due to these changes potentially going unnoticed.

Task 4. Complete the `CapacityOptimizer` class.

Implement the algorithm of Figure 1. Your implementation will go into the `getOptimalNumberOfSpots(...)` method of `CapacityOptimizer`. This method writes out statistics to the standard output. The format of the reported statistics is straightforward and illustrated below. Note that in our illustration, the results for lot sizes between 2 and 13 have been removed due to space.

```
$ java CapacityOptimizer 3
==== Setting lot capacity to: 1====
Simulation run 1 (23ms); Queue length at the end of simulation run: 68
Simulation run 2 (11ms); Queue length at the end of simulation run: 66
Simulation run 3 (11ms); Queue length at the end of simulation run: 73
Simulation run 4 (10ms); Queue length at the end of simulation run: 61
Simulation run 5 (10ms); Queue length at the end of simulation run: 82
Simulation run 6 (9ms); Queue length at the end of simulation run: 61
Simulation run 7 (10ms); Queue length at the end of simulation run: 76
Simulation run 8 (9ms); Queue length at the end of simulation run: 63
Simulation run 9 (8ms); Queue length at the end of simulation run: 61
Simulation run 10 (9ms); Queue length at the end of simulation run: 64

 [... RESULTS FOR LOT SIZES 2 to 13 NOT SHOWN DUE TO SPACE ...]

==== Setting lot capacity to: 14====
Simulation run 1 (92ms); Queue length at the end of simulation run: 1
Simulation run 2 (82ms); Queue length at the end of simulation run: 7
Simulation run 3 (103ms); Queue length at the end of simulation run: 4
Simulation run 4 (97ms); Queue length at the end of simulation run: 18
Simulation run 5 (95ms); Queue length at the end of simulation run: 14
Simulation run 6 (94ms); Queue length at the end of simulation run: 1
Simulation run 7 (98ms); Queue length at the end of simulation run: 4
Simulation run 8 (81ms); Queue length at the end of simulation run: 2
Simulation run 9 (102ms); Queue length at the end of simulation run: 8
Simulation run 10 (97ms); Queue length at the end of simulation run: 19

==== Setting lot capacity to: 15====
Simulation run 1 (95ms); Queue length at the end of simulation run: 4
Simulation run 2 (105ms); Queue length at the end of simulation run: 11
Simulation run 3 (95ms); Queue length at the end of simulation run: 0
Simulation run 4 (103ms); Queue length at the end of simulation run: 11
Simulation run 5 (92ms); Queue length at the end of simulation run: 10
Simulation run 6 (96ms); Queue length at the end of simulation run: 7
Simulation run 7 (90ms); Queue length at the end of simulation run: 0
Simulation run 8 (95ms); Queue length at the end of simulation run: 0
Simulation run 9 (99ms); Queue length at the end of simulation run: 2
Simulation run 10 (94ms); Queue length at the end of simulation run: 0

SIMULATION IS COMPLETE!
The smallest number of parking spots required: 15
Total execution time: 8.603 seconds
$
```

Alongside the A3 starter code, you will find several examples of output; see the `outputs` directory. **Note that due to the probabilistic nature of the simulation, you should not expect the same output when you simulate the same hourly rate a number of times.** What you will find to be more robust than individual simulation runs is the average across 10 runs. For example, if you set the arrival rate to 3 cars per hour, you are very likely to converge on 15 ± 2 as the required number of spots for the parking lot.

Task 5. Enhance Your Code with Exception Handling.

Implement exception handling for the methods in the following there classes: `ParkingLot`, `Simulator` and `TriangularDistribution`. Specifically, and for the purposes of this assignment, you are asked to check the validity of the arguments provided to the public methods for these three classes and throw suitable exceptions when invalid arguments are provided, e.g., an out-of-bound index or a null argument when a non-null argument is expected.

Implementation

We are now ready to program our solution. Just like in A2, you need to follow the patterns provided to you in the template code. For A3, you are **strongly discouraged from introducing any new (private) instance variables**. Introducing such variables can potentially take you away from the intended solution strategy and result in lost marks. **Notably, the occupancy information in `ParkingLot` must be implemented through lists and *not* arrays.**

You can complete the tasks in any order you like. However, I encourage you to follow the order in which the tasks appear (that is, Task 1, then Task 2, and so on). Like in A2, guidance is provided through comments in the starter 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 A3

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 A3:

```
java CapacityOptimizer <hourly rate of arrival>

Example usage: java CapacityOptimizer 11
```

Please feel free to alter `CapacityOptimizer.main(...)` to test and debug your implementation. **However, when you submit A3, please make sure to include the original `main(...)` method provided in the starter code.**

Example Outputs

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

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 Brightspace.
- 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: **a3_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 3. The parts are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. The archive a3_3000000_3000001.zip (available on Brightspace under Assignment 3) 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).
- **CapacityOptimizer.java**
- Car.java (submit as-is without changes)
- **LinkedQueue.java**
- List.java (submit as-is without changes)
- **ParkingLot.java**
- Queue.java (submit as-is without changes)
- RandomGenerator.java (submit as-is without changes)
- Rational.java (submit as-is without changes)
- **Simulator.java**
- SinglyLinkedList.java (submit as-is without changes)
- Spot.java (submit as-is without changes)
- **StudentInfo.java** (Update the `display()` method)
- **TriangularDistribution.java** (add exception handling only)

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:

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

Last modified: February 28, 2024