CS2030 Project: Discrete Event Simulator

Tags & Categories

Tags:

Categories:

Related Tutorials

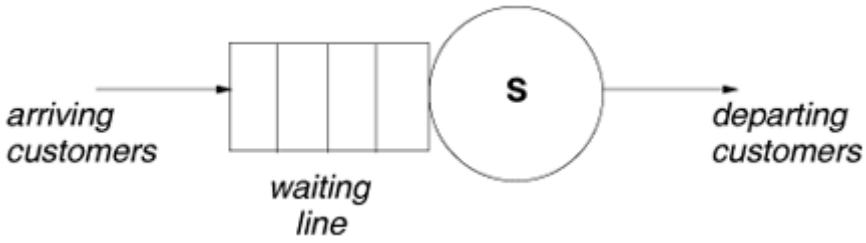
Task Content

CS2030 Project — Discrete Event Simulator

A discrete event simulator is a software that simulates the changes in the state of a system across time, with each transition from one state of the system to another triggered via an event. Such a system can be used to study many complex real-world systems such as queuing to order food at a fast-food restaurant.

An event occurs at a particular time, and each event alters the state of the system and may generate more events. States remain unchanged between two events (hence the term **discrete**), and this allows the simulator to jump from the time of one event to another. Some simulation statistics are also typically tracked to measure the performance of the system.

The following illustrates a queuing system comprising a single service point **S** with one customer queue.



In this exercise, we consider a multi-server system comprising the following:

- There are a number of **server**s; each server can serve one customer at a time.
- Each customer has a **service time** (time taken to serve the customer).
- When a **customer** arrives (`ARRIVE` event):
  - if the server is idle (not serving any customer), then the server starts serving the customer immediately (`SERVE` event).
  - if the server is serving another customer, then the customer that just arrived waits in the queue (`WAIT` event).
  - if the server is serving one customer with a second customer waiting in the queue, and a third customer arrives, then this latter customer leaves (`LEAVE` event). In other words, there is at most one customer waiting in the queue.
  - When the server is done serving a customer (`DONE` event), the server can start serving the customer waiting at the front of the queue (if any).
- If there is no waiting customer, then the server becomes idle again.

Notice from the above description that there are five events in the system, namely: `ARRIVE`, `SERVE`, `WAIT`, `LEAVE` and `DONE`. For each customer, these are the only possible event transitions:

- `ARRIVE` → `SERVE` → `DONE`
- `ARRIVE` → `WAIT` → `SERVE` → `DONE`
- `ARRIVE` → `LEAVE`

In essence, an event is tied to one customer. Depending on the current state of the system, triggering an event will result in the next state of the system, and possibly the next event. Events are also processed via a queue. At the start of the simulation, the queue only contains the customer arrival events. With every simulation time step, an event is retrieved from the queue to be processed, and any resulting event added to the queue. This process is repeated until there are no events left in the queue.

As an example, given the arrival times (represented as a floating point value for simplicity) of three customers and one server, and assuming a constant service time of `1.0`,

```
0.500
0.600
0.700
```

The simulation starts with three arrival events

```
<0.500 1 arrives>
<0.600 2 arrives>
<0.700 3 arrives>
```

The next event to pick is `<0.500 1 arrives>`. This schedules a `SERVE` event.

```
<0.500 1 serves by 1>
<0.600 2 arrives>
<0.700 3 arrives>
```

The next event to pick is `<0.500 1 served>`. This schedules a `DONE` event.

```
<0.600 2 arrives>
<0.700 3 arrives>
<1.500 1 done serving by 1>
```

The next event to pick is `<0.600 2 arrives>`...

This process is repeated until there are no more events.

Using our example, the entire simulation run results in the following output, each of the form `<time_event_occurred, customer_id, event_details>`

```
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 waits at 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 serves by 1
2.500 2 done serving by 1
```

Finally, statistics of the system that we need to keep track of are:

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.

In our example, the end-of-simulation statistics are respectively, `[0.450 2 1]`.

Priority Queuing

The main structure that is used to manage the events in any discrete event simulation is the Priority Queue. Java provides the `PriorityQueue` (a mutable class) that can be used to keep a collection of elements, where each element is given a certain priority.

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

Take Note

The requirements of this project will be released in stages. For each level `N`, you will be given a driver class `MainN` in the file `MainN.java`, and you will be required to create the `SimulateN` class to go along with it.

All utility classes are to be packaged in `cs2030.util`, while the rest of the classes related to the simulation are to be packaged in `cs2030.simulator`, with `SimulateN` as the only `public` facing classes.

The Pair and ImList classes are provided.

Compile your code using

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

and run your program to ensure that everything is still in good working order before proceeding to the next level.

Level 0

Before delving into the implementation of the simulator, you will need to come up with your own generic immutable priority queue, PQ<T>. Specifically, you will need to construct the following:

- A constructor that takes in a Comparator and creates an empty PQ<T>
- A method add that takes in an element of type T and returns the resulting PQ
- A method poll that takes no arguments and removes the highest priority element in the queue (as determined by the Comparator). Note that both the element and the resulting queue should be removed, i.e. the return type should be Pair<T, PQ<T>>
- A method isEmpty() that returns true if the PQ is empty, or false otherwise
- A toString method that returns a string representation of the PQ

As much as possible, you should adopt the *immutable delegation pattern* for your implementation. Also note that the Pair class has been provided for you.

```
jshell> import cs2030.util.*

jshell> Comparator<String> cmp = (x, y) -> x.compareTo(y)
cmp ==> $Lambda$15/0x00000008000a9440@7dc7cbad

jshell> PQ<String> pq = new PQ<String>(cmp)
pq ==> []

jshell> pq.add("one").add("two").add("three")
$.. ==> [one, two, three]

jshell> pq
pq ==> []

jshell> pq.isEmpty()
$.. ==> true

jshell> pq = pq.add("one").add("two").add("three")
pq ==> [one, two, three]

jshell> pq.poll()
$.. ==> (one, [three, two])

jshell> pq
pq ==> [one, two, three]

jshell> pq.isEmpty()
$.. ==> false

jshell> pq.poll().first()
$.. ==> "one"

jshell> pq.poll().second().poll().first()
$.. ==> "three"

jshell> pq = pq.poll().second().poll().second().poll().second()
pq ==> []

jshell> pq.isEmpty()
$.. ==> true

jshell> Comparator<Object> c = (x, y) -> x.hashCode() - y.hashCode()
c ==> $Lambda$22/0x00000008000b6040@464bee09

jshell> pq = new PQ<String>(c)
pq ==> []

jshell> pq.add("one").poll()
$.. ==> (one, [])
```

Note that the `PQ` class should be packaged in `cs2030.util`, together with the `Pair` class.

Level 1

You will start by building a *walking skeleton* of the discrete event simulator. With this framework in place, you would be able to flesh out the details according to the requirement in subsequent levels. All the classes should reside in the package `cs2030.simulator`.

Write an *immutable* `Customer` class to represent each arriving customer that is tagged with a customer ID (of type `int`), as well as an arrival time (of type `double`).

```
jshell> import cs2030.util.*

jshell> import cs2030.simulator.*

jshell> new Customer(1, 0.5)
$.. ==> 1

jshell> new Customer(2, 0.6)
$.. ==> 2

jshell> new Customer(3, 0.7)
$.. ==> 3
```

An event (i.e. `ARRIVE`, `SERVE`, `DONE`, `WAIT`, or `LEAVE`) is associated with a customer and an event time. Events are placed in a priority queue so that they can be polled one after another according to the time of occurrence of the event. Let us first create a dummy event `EventStub` to test that events are processed correctly using the priority queue, `PQ<Event>`.

In the `EventStub` class, include the constructor that takes in a customer and the event time. Also include an appropriate `toString` method to return the event time as a string.

You will also need to define an appropriate `EventComparator` when creating the priority queue.

```
jshell> Event event = new EventStub(new Customer(1, 0.5), 3.7) // event time of 3.7
event ==> 3.700

jshell> Comparator<Event> cmp = new EventComparator()
cmp ==> EventComparator@28d25987

jshell> PQ<Event> pq = new PQ<Event>(cmp)
pq ==> []

jshell> pq = pq.add(new EventStub(new Customer(1, 0.5), 3.5))
pq ==> [3.500]

jshell> pq = pq.add(new EventStub(new Customer(2, 0.6), 2.6))
pq ==> [2.600, 3.500]

jshell> pq = pq.add(new EventStub(new Customer(3, 0.7), 1.7))
pq ==> [1.700, 3.500, 2.600]

jshell> pq.poll().first()
$.. ==> 1.700

jshell> pq.poll().second().poll().first()
$.. ==> 2.600

jshell> pq.poll().second().poll().second().poll().first()
$.. ==> 3.500

jshell> pq.poll().second().poll().second().poll().second().isEmpty()
$.. ==> true
```

Level 2

Write an *immutable* `Server` class to represent a server that is tagged with a server ID (of type `int`). The servers provide their services within a shop. As such, write a `Shop` class that takes in a list of servers.

```
jshell> import cs2030.util.*

jshell> import cs2030.simulator.*

jshell> new Server(1)
```

```
$.. ==> 1

jshell> new Server(2)
$.. ==> 2

jshell> new Shop(List.<Server>of(new Server(1), new Server(2)))
$.. ==> [1, 2]
```

Now how do we link the customer(s) to the shop (or list of servers)? By using events! Every time an event is polled from the priority queue, the event (e.g. `SERVE` event) looks at the current state of the shop, and updates the state (e.g. an available server becomes busy) as well as to generate the next event (e.g. generating a `DONE` event to be added to the priority queue).

An `Event` interface is given below; you may modify it to suit your needs later.

```
package cs2030.simulator;

import java.util.Optional;
import cs2030.util.Pair;

interface Event {
    Pair<Optional<Event>, Shop> execute(Shop shop);
}
```

The `execute` method takes the current status of the shop, makes changes to the state of the shop based on the event, and returns the next event (if any) and updated shop status as a pair of values.

Now, include the `execute` method in the `EventStub` that takes in a `Shop`, and returns no event (represented by an empty optional) and the same shop as a pair.

```
jshell> Shop shop = new Shop(List.<Server>of(new Server(1), new Server(2)))
shop ==> [1, 2]

jshell> new EventStub(new Customer(1, 0.5), 3.5)
$.. ==> 3.500

jshell> new EventStub(new Customer(1, 0.5), 3.5).execute(shop)
$.. ==> (Optional.empty, [1, 2])
```

Since events are associated with customers and the shop associated with the servers, a particular instance of the simulation can be completely defined by the priority queue of events, as well as the status of the shop. Write the class `Simulate2` with a constructor that takes in the number of servers, together with a `List<Double>` of customer arrival times, and constructs a priority queue of `EventStub` events using the arrival time as the event time, and the status of the shop.

Include a `run` method within the `Simulator2` class that takes no arguments, such that invoking the method will cause the simulation to start by having the events in the priority queue polled one after another until the queue is empty. The `run` method should return a string.

```
jshell> new Simulate2(1, List.<Double>of(0.5, 0.6, 0.7))
$.. ==> Queue: [0.500, 0.600, 0.700]; Shop: [1]

jshell> System.out.println(new Simulate2(1, List.<Double>of(0.5, 0.6, 0.7)).run())
0.500
0.600
0.700
-- End of Simulation --

jshell> new Simulate2(2, List.<Double>of(3.5, 2.6, 1.7))
$.. ==> Queue: [1.700, 3.500, 2.600]; Shop: [1, 2]

jshell> System.out.println(new Simulate2(1, List.<Double>of(3.5, 2.6, 1.7)).run())
1.700
2.600
3.500
-- End of Simulation --
```

Use the program Main2.java provided to test your program as shown below. The first input value is the number of servers. This is followed by a sequence of input arrival times associated with each `Customer` having an identifier `1`, `2`, `3`, ... respectively. User input is underlined. Input is terminated with a `^D` (CTRL-d).

Take note of the following assumptions:

- The format of the input is always correct;

- Output of a `double` value, say `d`, is to be formatted with `String.format("%.3f", d);`

```
$ java Main2
1
0.500
0.600
0.700
^D
0.500
0.600
0.700
-- End of Simulation --
```

Level 3

If you have designed the simulator correctly, this level would only require you to construct the rest of the `ARRIVE`, `SERVE`, `DONE`, `WAIT` and `LEAVE` events in order to perform a proper simulation. For each event, you will need to define an appropriate `execute` method that takes in the current state of the shop, and return the appropriate next event, as well as the updated shop as a pair of values.

Using an `ARRIVE` event at time `t` as an example, if the `execute` method takes in a shop with all servers free, then the shop will return a serve event (to be served by `1` at time `t`), and the shop (no updates are necessary since service has not started). On the other hand, if the event was a `SERVE` event at time `t` at `i`, then the `execute` method takes in the shop, and returns a `DONE` event at time `t + 1.0` (assuming service time of `1.0`) together with the updated shop with server `i` tagged as busy.

From this point on, you will not be guided with `jshell` tests. Instead we will only provide sample runs using the provided `Main` programs.

Here's a tip: rather than creating all five events at the same time, you can still develop and test one event at a time. For example, start with the arrival event, and within it's `execute` method, return the `EventStub` with a later event time. At the same time, you can check the status of the shop to see if the updates are intended. Eventually, all arrival events will be processed and only `EventStub` events are left in the queue, which are then processed as per the earlier level. Once tested, you can then proceed to let arrive event's `execute` return serve event, and have serve event's `execute` method return `EventStub`.

Use the program `Main3.java` provided to test your program as shown below. User input is underlined, and starts with an integer value representing the number of servers, followed by the arrival times of the customers. Input is terminated with a `^D` (CTRL-d).

```
$ java Main3
1
0.500
0.600
0.700
^D
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 waits at 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 serves by 1
2.500 2 done serving by 1
-- End of Simulation --
```

```
$ java Main3
1
0.500
0.600
0.700
1.500
1.600
1.700
^D
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 waits at 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 serves by 1
```

```
1.500 4 arrives
1.500 4 waits at 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 serves by 1
3.500 4 done serving by 1
-- End of Simulation --
```

```
$ java Main3
2
0.500
0.600
0.700
1.500
1.600
1.700
^D
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 serves by 2
0.700 3 arrives
0.700 3 waits at 1
1.500 1 done serving by 1
1.500 3 serves by 1
1.500 4 arrives
1.500 4 waits at 1
1.600 2 done serving by 2
1.600 5 arrives
1.600 5 serves by 2
1.700 6 arrives
1.700 6 waits at 2
2.500 3 done serving by 1
2.500 4 serves by 1
2.600 5 done serving by 2
2.600 6 serves by 2
3.500 4 done serving by 1
3.600 6 done serving by 2
-- End of Simulation --
```

Level 4

You will also need to compute the statistics (see problem description above) at the end of the simulation.

Rather than changing the implementation details of the existing programs, try to create a `Statistic` object, and update this object as the events are polled from the priority queue.

Use the program Main4.java provided to test your program as shown below. User input is underlined, and starts with an integer value representing the number of servers, followed by the arrival times of the customers. Input is terminated with a `^D` (CTRL-d).

```
$ java Main4
1
0.500
0.600
0.700
^D
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 waits at 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 serves by 1
2.500 2 done serving by 1
[0.450 2 1]
```

```
$ java Main4
1
0.500
0.600
```

```
0.700
1.500
1.600
1.700
^D
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 waits at 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 serves by 1
1.500 4 arrives
1.500 4 waits at 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 serves by 1
3.500 4 done serving by 1
[0.633 3 3]
```

```
$ java Main4
2
0.500
0.600
0.700
1.500
1.600
1.700
^D
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 serves by 2
0.700 3 arrives
0.700 3 waits at 1
1.500 1 done serving by 1
1.500 3 serves by 1
1.500 4 arrives
1.500 4 waits at 1
1.600 2 done serving by 2
1.600 5 arrives
1.600 5 serves by 2
1.700 6 arrives
1.700 6 waits at 2
2.500 3 done serving by 1
2.500 4 serves by 1
2.600 5 done serving by 2
2.600 6 serves by 2
3.500 4 done serving by 1
3.600 6 done serving by 2
[0.450 6 0]
```

Level 5

Rather than a constant service time, each customer now has his/her own service time. As part of reading input, it would be desired that the arrival time and service time of each customer to be read as a pair of double values of type `Pair<Double, Double>`.

However, a customer might not be served and just leave, or the service time of the customer is not known until he/she is being served. In order to cater to such situations, the pair of values would be of type `Pair<Double, Supplier<Pair>>` instead where the first value of the pair is the arrival time that is read at the start, and the second value of the pair is the service time that is read only when it is needed.

We illustrate with an example below. Suppose a `Scanner` object is created to read arrival and service times for three customers.

```
jshell> Scanner sc = new Scanner(System.in)
sc ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
```

We create three pairs of values of type `Pair<Double, Supplier<Double>>` and store it in a `List`.

```
jshell> List<Pair<Double, Supplier<Double>>> list = Stream.<Scanner>generate(() -> sc).
   ...> limit(3).
   ...> map(x -> Pair.<Double, Supplier<Double>>of(sc.nextDouble(), () -> sc.nextDouble())).
   ...> collect(Collectors.toList())
0.5
0.6
0.7
list ==> [(0.5, $Lambda$35/0x00000008000d0840@bd8db5a), (0 ... 00000008000d0840@bd8db5a)]
```

Notice that each element in the list is a pair comprising a double value that has been read as input by `Scanner sc`, and a `Supplier<Double>` of the form `() -> sc.nextDouble()` where input is yet to be read.

```
jshell> list.get(0)
$.. ==> (0.5, $Lambda$35/0x00000008000d0840@bd8db5a)

jshell> list.get(1)
$.. ==> (0.6, $Lambda$35/0x00000008000d0840@bd8db5a)

jshell> list.get(2)
$.. ==> (0.7, $Lambda$35/0x00000008000d0840@bd8db5a)
```

Clearly, the three input are associated with the first value of each pair in the list.

```
jshell> list.get(0).first()
$.. ==> 0.5

jshell> list.get(1).first()
$.. ==> 0.6

jshell> list.get(2).first()
$.. ==> 0.7
```

The second value of each pair will not read input until the `Supplier`'s `get` method is invoked. For example,

```
jshell> list.get(1).second()
$.. ==> $Lambda$35/0x00000008000d0840@bd8db5a

jshell> list.get(1).second().get()
1.0
$.. ==> 1.0
```

What is interesting in the above is that the second customer with the arrival time of `0.6`, is the first one to be served with a new input of `1.0`. Also note that if you invoke the same jshell test again, another input is read and returned, rather than returning the value that was last read.

```
jshell> list.get(1).second().get() // same test
2.0
$.. ==> 2.0
```

Use the program `Main5.java` provided to test your program. The `Lazy` class in `Lazy.java` is also provided; use it if you need to cache the service time of the customer.

User input starts with an integer value representing the number of servers, and an integer value representing the number of customers. This is followed by the arrival times of the customers. Lastly, a number of service times (could be more than necessary) are provided.

Moreover, rather than typing out the user input, it would be more convenient if you save the input in a file, say `test5_1.in` and pipe the contents of the file or redirect the file as part of user input.

- To pipe the contents of the file, use: `cat test5_1.in | java Main5`
- To redirect the file, use: `java Main5 < test5_1.in`

```
$ cat test5_1.in
1 3
0.500
0.600
0.700
1.0
1.0
1.0
$ cat test5_1.in | java Main5
0.500 1 arrives
```

```
0.500 1 serves by 1
0.600 2 arrives
0.600 2 waits at 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 serves by 1
2.500 2 done serving by 1
[0.450 2 1]
```

```
$ cat test5_2.in
1 6
0.500
0.600
0.700
1.500
1.600
1.700
1.0
1.0
1.0
$ cat test5_2.in | java Main5
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 waits at 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 serves by 1
1.500 4 arrives
1.500 4 waits at 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 serves by 1
3.500 4 done serving by 1
[0.633 3 3]
```

```
$ cat test5_3.in
2 10
0.000000
0.313508
1.204910
2.776499
3.876961
3.909737
9.006391
9.043361
9.105379
9.159630
0.313141
0.103753
0.699522
0.014224
0.154173
0.012659
1.477717
2.593020
0.296847
0.051732
$ cat test5_3.in | java Main5
0.000 1 arrives
0.000 1 serves by 1
0.313 1 done serving by 1
0.314 2 arrives
0.314 2 serves by 1
0.417 2 done serving by 1
1.205 3 arrives
1.205 3 serves by 1
1.904 3 done serving by 1
```

```
2.776 4 arrives
2.776 4 serves by 1
2.791 4 done serving by 1
3.877 5 arrives
3.877 5 serves by 1
3.910 6 arrives
3.910 6 serves by 2
3.922 6 done serving by 2
4.031 5 done serving by 1
9.006 7 arrives
9.006 7 serves by 1
9.043 8 arrives
9.043 8 serves by 2
9.105 9 arrives
9.105 9 waits at 1
9.160 10 arrives
9.160 10 waits at 2
10.484 7 done serving by 1
10.484 9 serves by 1
10.781 9 done serving by 1
11.636 8 done serving by 2
11.636 10 serves by 2
11.688 10 done serving by 2
[0.386 10 0]
```

```
$ cat test5_4.in
2 10
0.000000
0.313141
0.416894
1.116416
1.130640
1.284813
1.297472
2.775189
5.368209
5.665056
0.313263
2.645188
2.701273
0.263761
1.481332
0.415031
0.214836
$ cat test5_4.in | java Main5
0.000 1 arrives
0.000 1 serves by 1
0.313 2 arrives
0.313 2 serves by 2
0.313 1 done serving by 1
0.417 3 arrives
0.417 3 serves by 1
1.116 4 arrives
1.116 4 waits at 1
1.131 5 arrives
1.131 5 waits at 2
1.285 6 arrives
1.285 6 leaves
1.297 7 arrives
1.297 7 leaves
2.775 8 arrives
2.775 8 leaves
2.958 2 done serving by 2
2.958 5 serves by 2
3.118 3 done serving by 1
3.118 4 serves by 1
3.222 5 done serving by 2
4.599 4 done serving by 1
5.368 9 arrives
5.368 9 serves by 1
5.665 10 arrives
5.665 10 serves by 2
5.783 9 done serving by 1
```

```
5.880 10 done serving by 2
[0.547 7 3]
```

Level 6

Each server now has a queue of customers to allow multiple customers to queue up. A customer that chooses the first queue available and joins at the tail. When a server is done serving a customer, it serves the next waiting customer at the head of the queue. Hence, the queue should be a first-in-first-out (FIFO) structure.

Use the program Main6.java provided to test your program.

User input starts with three integer values representing the number of servers, followed by the maximum queue length, and the number of customers. This is followed by the arrival times of the customers. Lastly, a number of service times (could be more than necessary) are provided.

```
$ cat test6_1.in
2 1 10
0.000000
0.313508
1.204910
2.776499
3.876961
3.909737
9.006391
9.043361
9.105379
9.159630
0.313141
0.103753
0.699522
0.014224
0.154173
0.012659
1.477717
2.593020
0.296847
0.051732
$ cat test6_1.in | java Main6
0.000 1 arrives
0.000 1 serves by 1
0.313 1 done serving by 1
0.314 2 arrives
0.314 2 serves by 1
0.417 2 done serving by 1
1.205 3 arrives
1.205 3 serves by 1
1.904 3 done serving by 1
2.776 4 arrives
2.776 4 serves by 1
2.791 4 done serving by 1
3.877 5 arrives
3.877 5 serves by 1
3.910 6 arrives
3.910 6 serves by 2
3.922 6 done serving by 2
4.031 5 done serving by 1
9.006 7 arrives
9.006 7 serves by 1
9.043 8 arrives
9.043 8 serves by 2
9.105 9 arrives
9.105 9 waits at 1
9.160 10 arrives
9.160 10 waits at 2
10.484 7 done serving by 1
10.484 9 serves by 1
10.781 9 done serving by 1
11.636 8 done serving by 2
11.636 10 serves by 2
11.688 10 done serving by 2
[0.386 10 0]
```

```
$ cat test6_2.in
2 2 10
0.000000
0.313508
1.204910
2.776499
3.876961
3.909737
9.006391
9.043361
9.105379
9.159630
0.313141
0.103753
0.699522
0.014224
0.154173
0.012659
1.477717
2.593020
0.296847
0.051732
$ cat test6_2.in | java Main6
0.000 1 arrives
0.000 1 serves by 1
0.313 1 done serving by 1
0.314 2 arrives
0.314 2 serves by 1
0.417 2 done serving by 1
1.205 3 arrives
1.205 3 serves by 1
1.904 3 done serving by 1
2.776 4 arrives
2.776 4 serves by 1
2.791 4 done serving by 1
3.877 5 arrives
3.877 5 serves by 1
3.910 6 arrives
3.910 6 serves by 2
3.922 6 done serving by 2
4.031 5 done serving by 1
9.006 7 arrives
9.006 7 serves by 1
9.043 8 arrives
9.043 8 serves by 2
9.105 9 arrives
9.105 9 waits at 1
9.160 10 arrives
9.160 10 waits at 1
10.484 7 done serving by 1
10.484 9 serves by 1
10.781 9 done serving by 1
10.781 10 serves by 1
10.833 10 done serving by 1
11.636 8 done serving by 2
[0.300 10 0]
```

```
$ cat test6_3.in
2 2 10
0.000000
0.313141
0.416894
1.116416
1.130640
1.284813
1.297472
2.775189
5.368209
5.665056
0.313263
2.645188
2.701273
0.263761
```

```
1.481332
0.415031
0.214836
3.512654
0.209277
$ cat test6_3.in | java Main6
0.000 1 arrives
0.000 1 serves by 1
0.313 2 arrives
0.313 2 serves by 2
0.313 1 done serving by 1
0.417 3 arrives
0.417 3 serves by 1
1.116 4 arrives
1.116 4 waits at 1
1.131 5 arrives
1.131 5 waits at 1
1.285 6 arrives
1.285 6 waits at 2
1.297 7 arrives
1.297 7 waits at 2
2.775 8 arrives
2.775 8 leaves
2.958 2 done serving by 2
2.958 6 serves by 2
3.118 3 done serving by 1
3.118 4 serves by 1
3.222 6 done serving by 2
3.222 7 serves by 2
3.637 7 done serving by 2
4.599 4 done serving by 1
4.599 5 serves by 1
4.814 5 done serving by 1
5.368 9 arrives
5.368 9 serves by 1
5.665 10 arrives
5.665 10 serves by 2
5.874 10 done serving by 2
8.881 9 done serving by 1
[1.008 9 1]
```

Level 7

The servers are now allowed to take occasional breaks. When a server finishes serving a customer, there is a chance that the server takes a rest for a certain amount of time. During the break, the server does not serve the next waiting customer. Upon returning from the break, the server serves the next customer in the queue (if any) immediately.

Just like customer service time, we cannot pre-determine what rest time is accorded to which server at the beginning; it can only be decided during the simulation. That is to say, whenever a server rests, it will then know how much time it has to rest. The service time is supplied by a random number generator as shown in the program Main7.java provided.

User input starts with values representing the number of servers, followed by the maximum queue length, the number of customers and the probability of a server resting. This is followed by the arrival times of the customers. Lastly, a number of service times (could be more than necessary) are provided.

Two sample runs are shown below: the first with no rest, and the second with probability of rest set to 0.5. In this latter case, after server 1 is done serving customer 2 at 0.417, he/she takes a rest. In the meantime, customers 3 and 4 are served by server 2. Server 1 resumes serving customer 5 after resting.

```
$ cat 7_1.in
2 2 10 0
0.000000
0.313508
1.204910
2.776499
3.876961
3.909737
9.006391
9.043361
9.105379
9.159630
0.313141
0.103753
0.699522
```

```
0.014224
0.154173
0.012659
1.477717
2.593020
0.296847
0.051732

$ cat 7_1.in | java Main7
0.000 1 arrives
0.000 1 serves by 1
0.313 1 done serving by 1
0.314 2 arrives
0.314 2 serves by 1
0.417 2 done serving by 1
1.205 3 arrives
1.205 3 serves by 1
1.904 3 done serving by 1
2.776 4 arrives
2.776 4 serves by 1
2.791 4 done serving by 1
3.877 5 arrives
3.877 5 serves by 1
3.910 6 arrives
3.910 6 serves by 2
3.922 6 done serving by 2
4.031 5 done serving by 1
9.006 7 arrives
9.006 7 serves by 1
9.043 8 arrives
9.043 8 serves by 2
9.105 9 arrives
9.105 9 waits at 1
9.160 10 arrives
9.160 10 waits at 1
10.484 7 done serving by 1
10.484 9 serves by 1
10.781 9 done serving by 1
10.781 10 serves by 1
10.833 10 done serving by 1
11.636 8 done serving by 2
[0.300 10 0]
```

```
$ cat 7_2.in
2 2 10 0.5
0.000000
0.313508
1.204910
2.776499
3.876961
3.909737
9.006391
9.043361
9.105379
9.159630
0.313141
0.103753
0.699522
0.014224
0.154173
0.012659
1.477717
2.593020
0.296847
0.051732

$ cat 7_2.in | java Main7
0.000 1 arrives
0.000 1 serves by 1
0.313 1 done serving by 1
0.314 2 arrives
0.314 2 serves by 1
0.417 2 done serving by 1
1.205 3 arrives
```

```
1.205 3 serves by 2
1.904 3 done serving by 2
2.776 4 arrives
2.776 4 serves by 2
2.791 4 done serving by 2
3.877 5 arrives
3.877 5 serves by 1
3.910 6 arrives
3.910 6 serves by 2
3.922 6 done serving by 2
4.031 5 done serving by 1
9.006 7 arrives
9.006 7 serves by 1
9.043 8 arrives
9.043 8 serves by 2
9.105 9 arrives
9.105 9 waits at 1
9.160 10 arrives
9.160 10 waits at 1
10.484 7 done serving by 1
10.484 9 serves by 1
10.781 9 done serving by 1
11.636 8 done serving by 2
14.644 10 serves by 1
14.696 10 done serving by 1
[0.686 10 0]
```

Level 8

There are now N_{self} self-checkout counters set up. In particular, if there are k human servers, then the self-checkout counters are identified from $k + 1$ onwards.

Take note of the following:

- All self-checkout counters share the same queue.
- Unlike human servers, self-checkout counters do not rest.
- When we print out the wait event, we always say that the customer is waiting for the self-checkout counter $k + 1$, even though this customer may eventually be served by another self-checkout counter.

Use the program Main8.java provided to test your program.

User input starts with values representing the number of servers, the number of self-check counters, the maximum queue length, the number of customers and the probability of a server resting. This is followed by the arrival times of the customers. Lastly, a number of service times (could be more than necessary) are provided.

```
$ cat 8_1.in
2 0 2 20 0.5
0.000000
0.313508
1.204910
2.776499
3.876961
3.909737
9.006391
9.043361
9.105379
9.159630
9.224614
10.147994
11.204933
12.428914
13.109178
15.263626
15.524296
15.939974
17.793096
18.765423
0.313141
0.103753
0.699522
0.014224
0.154173
0.012659
1.477717
```

```
2.593020
0.296847
0.051732
3.489595
0.368667
0.160532
2.869150
0.895790
1.043020
0.006333
0.576351
0.742370

$ cat 8_1.in | java Main8
0.000 1 arrives
0.000 1 serves by 1
0.313 1 done serving by 1
0.314 2 arrives
0.314 2 serves by 1
0.417 2 done serving by 1
1.205 3 arrives
1.205 3 serves by 2
1.904 3 done serving by 2
2.776 4 arrives
2.776 4 serves by 2
2.791 4 done serving by 2
3.877 5 arrives
3.877 5 serves by 1
3.910 6 arrives
3.910 6 serves by 2
3.922 6 done serving by 2
4.031 5 done serving by 1
9.006 7 arrives
9.006 7 serves by 1
9.043 8 arrives
9.043 8 serves by 2
9.105 9 arrives
9.105 9 waits at 1
9.160 10 arrives
9.160 10 waits at 1
9.225 11 arrives
9.225 11 waits at 2
10.148 12 arrives
10.148 12 waits at 2
10.484 7 done serving by 1
10.484 9 serves by 1
10.781 9 done serving by 1
11.205 13 arrives
11.205 13 waits at 1
11.636 8 done serving by 2
11.636 11 serves by 2
11.688 11 done serving by 2
11.688 12 serves by 2
12.429 14 arrives
12.429 14 waits at 2
13.109 15 arrives
13.109 15 waits at 2
14.644 10 serves by 1
15.013 10 done serving by 1
15.178 12 done serving by 2
15.178 14 serves by 2
15.264 16 arrives
15.264 16 waits at 1
15.338 14 done serving by 2
15.338 15 serves by 2
15.524 17 arrives
15.524 17 waits at 2
15.940 18 arrives
15.940 18 waits at 2
17.793 19 arrives
17.793 19 leaves
18.207 15 done serving by 2
18.207 17 serves by 2
18.765 20 arrives
```

```
18.765 20 waits at 2
19.103 17 done serving by 2
19.103 18 serves by 2
20.146 18 done serving by 2
40.474 13 serves by 1
40.480 13 done serving by 1
40.480 16 serves by 1
41.056 16 done serving by 1
57.110 20 serves by 2
57.852 20 done serving by 2
[6.025 19 1]
```

```
$ cat 8_2.in
2 1 2 20 0.5
0.000000
0.313508
1.204910
2.776499
3.876961
3.909737
9.006391
9.043361
9.105379
9.159630
9.224614
10.147994
11.204933
12.428914
13.109178
15.263626
15.524296
15.939974
17.793096
18.765423
0.313141
0.103753
0.699522
0.014224
0.154173
0.012659
1.477717
2.593020
0.296847
0.051732
3.489595
0.368667
0.160532
2.869150
0.895790
1.043020
0.006333
0.576351
0.742370
3.007573

$ cat 8_2.in | java Main8
0.000 1 arrives
0.000 1 serves by 1
0.313 1 done serving by 1
0.314 2 arrives
0.314 2 serves by 1
0.417 2 done serving by 1
1.205 3 arrives
1.205 3 serves by 2
1.904 3 done serving by 2
2.776 4 arrives
2.776 4 serves by 2
2.791 4 done serving by 2
3.877 5 arrives
3.877 5 serves by 1
3.910 6 arrives
3.910 6 serves by 2
3.922 6 done serving by 2
4.031 5 done serving by 1
```

```
9.006 7 arrives
9.006 7 serves by 1
9.043 8 arrives
9.043 8 serves by 2
9.105 9 arrives
9.105 9 serves by self-check 3
9.160 10 arrives
9.160 10 waits at 1
9.225 11 arrives
9.225 11 waits at 1
9.402 9 done serving by self-check 3
10.148 12 arrives
10.148 12 serves by self-check 3
10.200 12 done serving by self-check 3
10.484 7 done serving by 1
10.484 10 serves by 1
11.205 13 arrives
11.205 13 serves by self-check 3
11.574 13 done serving by self-check 3
11.636 8 done serving by 2
12.429 14 arrives
12.429 14 serves by self-check 3
12.589 14 done serving by self-check 3
13.109 15 arrives
13.109 15 serves by self-check 3
13.974 10 done serving by 1
13.974 11 serves by 1
14.869 11 done serving by 1
15.264 16 arrives
15.264 16 serves by 1
15.524 17 arrives
15.524 17 serves by 2
15.531 17 done serving by 2
15.940 18 arrives
15.940 18 waits at 1
15.978 15 done serving by self-check 3
16.307 16 done serving by 1
16.307 18 serves by 1
16.883 18 done serving by 1
17.793 19 arrives
17.793 19 serves by 1
18.535 19 done serving by 1
18.765 20 arrives
18.765 20 serves by 1
21.773 20 done serving by 1
[0.322 20 0]
```

```
$ cat 8_3.in
1 2 2 20 0.5
0.000000
0.313508
1.204910
2.776499
3.876961
3.909737
9.006391
9.043361
9.105379
9.159630
9.224614
10.147994
11.204933
12.428914
13.109178
15.263626
15.524296
15.939974
17.793096
18.765423
0.313141
0.103753
0.699522
0.014224
0.154173
```

```
0.012659
1.477717
2.593020
0.296847
0.051732
3.489595
0.368667
0.160532
2.869150
0.895790
1.043020
0.006333
0.576351
0.742370
3.007573

$ cat 8_3.in | java Main8
0.000 1 arrives
0.000 1 serves by 1
0.313 1 done serving by 1
0.314 2 arrives
0.314 2 serves by 1
0.417 2 done serving by 1
1.205 3 arrives
1.205 3 serves by self-check 2
1.904 3 done serving by self-check 2
2.776 4 arrives
2.776 4 serves by self-check 2
2.791 4 done serving by self-check 2
3.877 5 arrives
3.877 5 serves by 1
3.910 6 arrives
3.910 6 serves by self-check 2
3.922 6 done serving by self-check 2
4.031 5 done serving by 1
9.006 7 arrives
9.006 7 serves by 1
9.043 8 arrives
9.043 8 serves by self-check 2
9.105 9 arrives
9.105 9 serves by self-check 3
9.160 10 arrives
9.160 10 waits at 1
9.225 11 arrives
9.225 11 waits at 1
9.402 9 done serving by self-check 3
10.148 12 arrives
10.148 12 serves by self-check 3
10.200 12 done serving by self-check 3
10.484 7 done serving by 1
10.484 10 serves by 1
11.205 13 arrives
11.205 13 serves by self-check 3
11.574 13 done serving by self-check 3
11.636 8 done serving by self-check 2
12.429 14 arrives
12.429 14 serves by self-check 2
12.589 14 done serving by self-check 2
13.109 15 arrives
13.109 15 serves by self-check 2
13.974 10 done serving by 1
14.823 11 serves by 1
15.264 16 arrives
15.264 16 serves by self-check 3
15.524 17 arrives
15.524 17 waits at 1
15.718 11 done serving by 1
15.718 17 serves by 1
15.725 17 done serving by 1
15.940 18 arrives
15.940 18 serves by 1
15.978 15 done serving by self-check 2
16.307 16 done serving by self-check 3
16.516 18 done serving by 1
```

```
17.793 19 arrives
17.793 19 serves by self-check 2
18.535 19 done serving by self-check 2
18.765 20 arrives
18.765 20 serves by self-check 2
21.773 20 done serving by self-check 2
[0.356 20 0]
```

```
$ cat 8_4.in
1 2 2 20 0.5
0.000000
0.313508
1.204910
2.776499
3.876961
3.909737
9.006391
9.043361
9.105379
9.159630
9.224614
10.147994
11.204933
12.428914
13.109178
15.263626
15.524296
15.939974
17.793096
18.765423
3.131408
1.037533
6.995223
0.142237
1.541726
0.126590
14.777172
25.930199
2.968470
0.517321
34.895950
3.686675
1.605316
28.691500
8.957896
1.043020
0.006333
0.576351
0.742370
3.007573

$ cat 8_4.in | java Main8
0.000 1 arrives
0.000 1 serves by 1
0.314 2 arrives
0.314 2 serves by self-check 2
1.205 3 arrives
1.205 3 serves by self-check 3
1.351 2 done serving by self-check 2
2.776 4 arrives
2.776 4 serves by self-check 2
2.919 4 done serving by self-check 2
3.131 1 done serving by 1
3.877 5 arrives
3.877 5 serves by 1
3.910 6 arrives
3.910 6 serves by self-check 2
4.036 6 done serving by self-check 2
5.419 5 done serving by 1
8.200 3 done serving by self-check 3
9.006 7 arrives
9.006 7 serves by 1
9.043 8 arrives
9.043 8 serves by self-check 2
```

```
9.105 9 arrives
9.105 9 serves by self-check 3
9.160 10 arrives
9.160 10 waits at 1
9.225 11 arrives
9.225 11 waits at 1
10.148 12 arrives
10.148 12 waits at self-check 2
11.205 13 arrives
11.205 13 waits at self-check 2
12.074 9 done serving by self-check 3
12.074 12 serves by self-check 3
12.429 14 arrives
12.429 14 waits at self-check 2
12.591 12 done serving by self-check 3
12.591 13 serves by self-check 3
13.109 15 arrives
13.109 15 waits at self-check 2
15.264 16 arrives
15.264 16 leaves
15.524 17 arrives
15.524 17 leaves
15.940 18 arrives
15.940 18 leaves
17.793 19 arrives
17.793 19 leaves
18.765 20 arrives
18.765 20 leaves
23.784 7 done serving by 1
24.631 10 serves by 1
28.318 10 done serving by 1
28.318 11 serves by 1
29.923 11 done serving by 1
34.974 8 done serving by self-check 2
34.974 14 serves by self-check 2
47.487 13 done serving by self-check 3
47.487 15 serves by self-check 3
56.445 15 done serving by self-check 3
63.665 14 done serving by self-check 2
[6.320 15 5]
```

Remaining level(s) for correctness/style checking...