Simulation

Chapter 13: Simulation

Simulation

- Simulation is a tool that has been commonly used to assist with systems analysis.
  - Develop a simulation program that emulates the operation of a real system
  - Run the simulation program and gather statistics

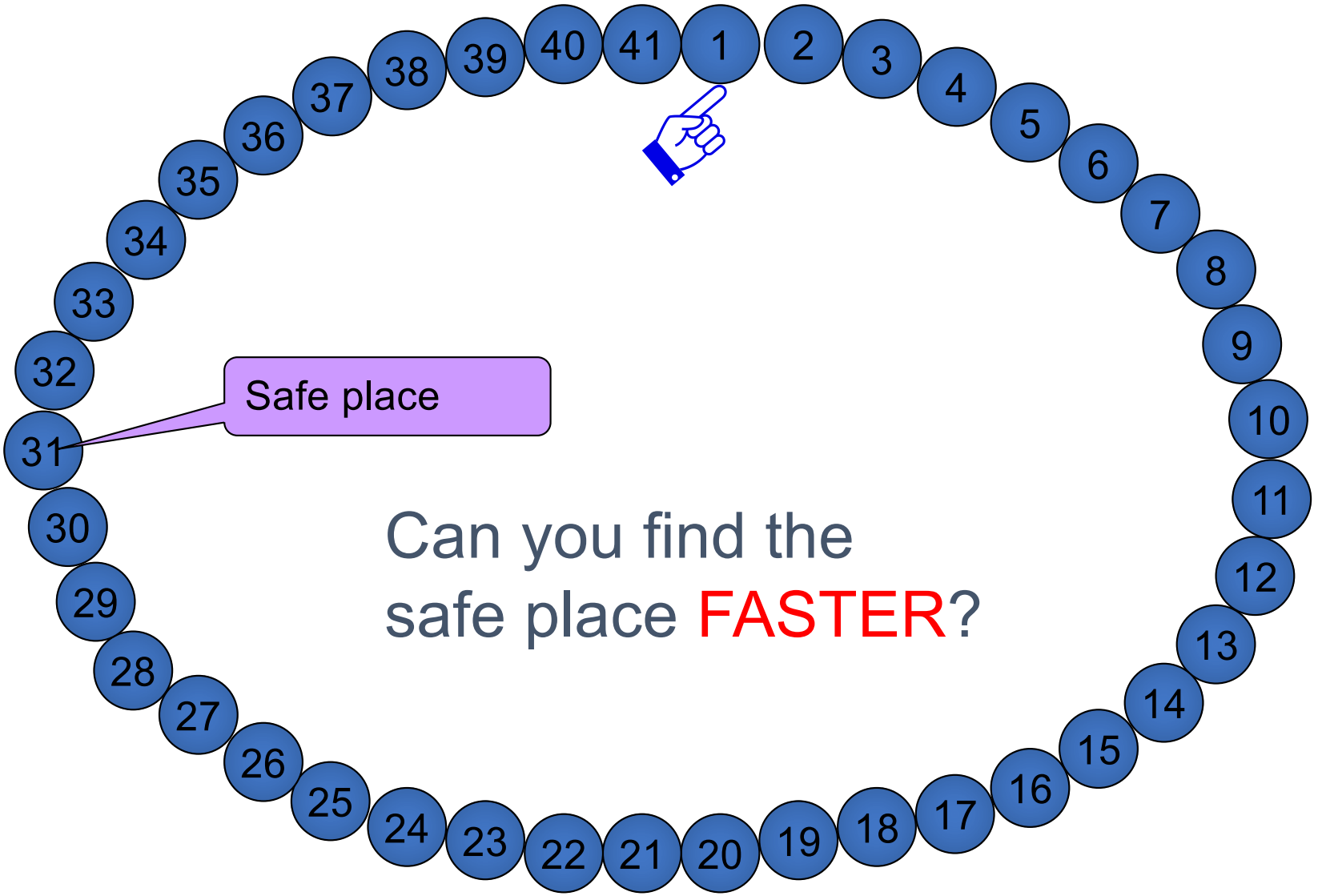
- Simulation models used for
  - Design analysis
  - Procedural analysis
  - Performance assessment

- Advantages:
  - Information would be gathered without involving real customers
  - Simulation by computer can be faster because of the speed of computer
  - Simulation could be easily replicated

Josephus Problem

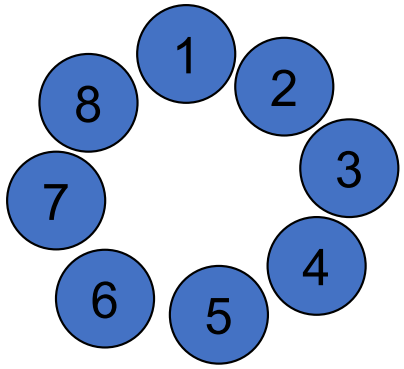
Flavius Josephus is a Jewish historian living in the 1st century. According to his account, he and his 40 comrade soldiers were trapped in a cave, surrounded by Romans. They chose suicide over capture and decided that they would form a circle and start killing one by skipping every two others. By luck, or maybe by the hand of God, Josephus and another man remained the last and gave up to the Romans.

Can You Find the Safe Place?



A Simpler Version

- Let's consider a similar problem
  - There are *n* person in a circle, numbered from 1 to *n* sequentially.
  - Starting from the number 1 person, every m^{th} person will be killed.
  - What is the safe place?
- The input is *n*, the output *f*(*n*) is a number between 1 and *n*.
  - Ex: f(8) = ?



Simple Simulation Solution

- We can find $f(n)$ using simulation.
  - We do not need to "kill" anyone to find $f(n)$.
- The simulation needs
  1) a **model** to represents "n people in a circle"
  2) a way to **simulate** "kill every m^{th} person"
  3) knowing when to stop

- How to perform simulations on computers?
  - A model of n-people-circle: (circular) linked list
  - Simulate "kill every m^{th} person": Remove every 2^{nd} node in the circular linked list
  - Knowing when to stop: The above process can loop until only one node left

- Running time: O(m*n)

figure 13.2

Linked list implementation of the Josephus problem

```
1       /**
2        * Return the winner in the Josephus problem.
3        * Linked list implementation.
4        * (Can replace with ArrayList or TreeSet).
5        */
6       public static int josephus( int people, int passes )
7       {
8           Collection<Integer> theList = new LinkedList<Integer>( );
9
10              // Construct the list
11          for( int i = 1; i <= people; i++ )
12              theList.add( i );
13
14              // Play the game;
15          Iterator<Integer> itr = theList.iterator( );
16          while( people-- != 1 )
17          {
18              for( int i = 0; i <= passes; i++ )
19              {
20                  if( !itr.hasNext( ) )
21                      itr = theList.iterator( );
22
23                  itr.next( );
24              }
25              itr.remove( );
26          }
27
28          itr = theList.iterator( );
29
30          return itr.next( );
31      }
```

In-Class Activity

- Download the Josephus.java file from CANVAS;
- Run the code;
- Read the code and try to find answers to the following questions:
  1) What is the **model** to represents “n people in a circle”?
  2) Which part is to **simulate** “kill every m^{th} person”?
  3) What is the stop criteria for the simulation?
  4) Try to add more comments to the main code in the block started with the comment “Play the game”.

Time-Driven vs. Event-Driven Simulation

- Time driven simulation
  - Use a variable recording the current time, which is incremented in fixed steps.
  - After each increment we check to see which events may happen at the current time point, and handle those that do
  - E.g.: simulate the trajectory of a projectile
- Event-driven simulation
  - If events aren't guaranteed to occur at regular intervals
  - Uses a list of events that occur at various time, and handles them in order of increasing time.
  - Handling an event may alter the list of later events.
  - The simulation makes time "jump" to the time of the next event.
  - E.g.: simulating a lineup at a bank, gas station, etc.

Generic Simulation Algorithms

- Time-driven:
  - Initialize the system state
  - Initialize simulation time
  - While (simulation is not finished)
    - Collect statistics about the current state
    - Handle events that occurred between last step and now
    - Increment simulation time
- Event-driven:
  - Initialize system state
  - Initialize event list
  - While (simulation not finished)
    - Collect statistics from current state
    - Remove first event from list, handle it
    - Set time to the time of this event.

Call Bank Problem

- A system: customers arrive and wait in line until one of K tellers is available
- Event of customer arrival is governed by a probability distribution function
  - As is the service time (the amount of time to be served once a teller becomes available)
- Statistics of interest:
  - How long on average a customer has to wait?
  - What percentage of tellers are actually servicing requests?

Basic Ideas

- Processing events
  - In bank simulation problem, events of customer arrival and customer departing
- Probability distribution function to generate input stream
  - In bank simulation problem, ordered pairs of arrival and service time for each customer, sorted by arrival time
- Simulation strategy
  - Simulation clock: A quantum unit, referred to as a tick
  - Starts at Tick 0, and advance the clock one tick at a time, checking to see if an event occurs. If so, process the events and compile statistics.
  - Key: advance the clock to the next event time at each stage
    - Find the event that happens soonest
    - Process the event (i.e. setting current time to the time that event occurs)
  - When no unserved event left, all resources are free, the simulation is over

Processing Events

- Customer departure event
  - Gather statistics for the departing customer and checking the line (queue) to determine whether another customer is waiting
    - If yes, add the customer, process whatever statistics are required, computer the time when the customer will leave, and add the departure to the set of events waiting to happen
- Customer arrival event
  - Check for available teller
    - If none, place the arrival in the line (queue)
    - Otherwise, assign the customer a teller, computer the customer's departure time, and add the departure to the set of events waiting to happen
- Waiting line can be implemented as a queue
  - Regular queue
  - If need to find the next soonest event: priority queue

Sample simulation log of call bank events

```
1  User 0 dials in at time 0 and connects for 1 minute
2  User 0 hangs up at time 1
3  User 1 dials in at time 1 and connects for 5 minutes
4  User 2 dials in at time 2 and connects for 4 minutes
5  User 3 dials in at time 3 and connects for 11 minutes
6  User 4 dials in at time 4 but gets busy signal
7  User 5 dials in at time 5 but gets busy signal
8  User 6 dials in at time 6 but gets busy signal
9  User 1 hangs up at time 6
10 User 2 hangs up at time 6
11 User 7 dials in at time 7 and connects for 8 minutes
12 User 8 dials in at time 8 and connects for 6 minutes
13 User 9 dials in at time 9 but gets busy signal
14 User 10 dials in at time 10 but gets busy signal
15 User 11 dials in at time 11 but gets busy signal
16 User 12 dials in at time 12 but gets busy signal
17 User 13 dials in at time 13 but gets busy signal
18 User 3 hangs up at time 14
19 User 14 dials in at time 14 and connects for 6 minutes
20 User 8 hangs up at time 14
21 User 15 dials in at time 15 and connects for 3 minutes
22 User 7 hangs up at time 15
23 User 16 dials in at time 16 and connects for 5 minutes
24 User 17 dials in at time 17 but gets busy signal
25 User 15 hangs up at time 18
26 User 18 dials in at time 18 and connects for 7 minutes
```

figure 13.4

Sample output for the modem bank simulation involving three modems: A dial-in is attempted every minute; the average connect time is 5 minutes; and the simulation is run for 18 minutes

figure 13.5

The Event class used for modem simulation

```
1   /**
2    * The event class.
3    * Implements the Comparable interface
4    * to arrange events by time of occurrence.
5    * (nested in ModemSim)
6    */
7   private static class Event implements Comparable<Event>
8   {
9       static final int DIAL_IN = 1;
10      static final int HANG_UP = 2;
11
12      public Event( )
13      {
14          this( 0, 0, DIAL_IN );
15      }
16
17      public Event( int name, int tm, int type )
18      {
19          who  = name;
20          time = tm;
21          what = type;
22      }
23
24      public int compareTo( Event rhs )
25      {
26          return time - rhs.time;
27      }
28
29      int who;        // the number of the user
30      int time;       // when the event will occur
31      int what;       // DIAL_IN or HANG_UP
32  }
```

```
1  import java.util.Random;
2  import java.util.PriorityQueue;
3
4  // ModemSim clas interface: run a simulation
5  //
6  // CONSTRUCTION: with three parameters: the number of
7  //     modems, the average connect time, and the
8  //     interarrival time
9  //
10 // ******************PUBLIC OPERATIONS*********************
11 // void runSim( )       --> Run a simulation
12
13 public class ModemSim
14 {
15     public ModemSim( int modems, double avgLen, int callIntrvl )
16       { /* Figure 13.7 */ }
17
18       // Run the simulation.
19     public void runSim( long stoppingTime )
20       { /* Figure 13.9 */ }
21
22       // Add a call to eventSet at the current time,
23       // and schedule one for delta in the future.
24     private void nextCall( int delta )
25       { /* Figure 13.8 */ }
26
27     private Random r;                        // A random source
28     private PriorityQueue<Event> eventSet;  // Pending events
29
30         // Basic parameters of the simulation
31     private int freeModems;           // Number of modems unused
32     private double avgCallLen;        // Length of a call
33     private int freqOfCalls;          // Interval between calls
34
35     private static class Event implements Comparable<Event>
36       { /* Figure 13.5 */ }
37 }
```

figure 13.6

The `ModemSim` class skeleton

figure 13.7

The `ModemSim` constructor

```
1     /**
2      * Constructor.
3      * @param modem number of modems.
4      * @param avgLen averge length of a call.
5      * @param callIntrvl the average time between calls.
6      */
7     public ModemSim( int modems, double avgLen, int callIntrvl )
8     {
9         eventSet    = new PriorityQueue<Event>( );
10        freeModems  = modems;
11        avgCallLen  = avgLen;
12        freqOfCalls = callIntrvl;
13        r           = new Random( );
14        nextCall( freqOfCalls );  // Schedule first call
15    }
```

figure 13.8

The `nextCall` method places a new `DIAL_IN` event in the event queue and advances the time when the next `DIAL_IN` event will occur

```
1       private int userNum = 0;
2       private int nextCallTime = 0;
3
4       /**
5        * Place a new DIAL_IN event into the event queue.
6        * Then advance the time when next DIAL_IN event will occur.
7        * In practice, we would use a random number to set the time.
8        */
9       private void nextCall( int delta )
10      {
11          Event ev = new Event( userNum++, nextCallTime, Event.DIAL_IN );
12          eventSet.insert( ev );
13          nextCallTime += delta;
14      }
```

```
1     /**
2      * Run the simulation until stoppingTime occurs.
3      * Print output as in Figure 13.4.
4      */
5     public void runSim( long stoppingTime )
6     {
7         Event e = null;
8         int howLong;
9
10        while( !eventSet.isEmpty( ) )
11        {
12            e = eventSet.remove( );
13
14            if( e.time > stoppingTime )
15                break;
16
17            if( e.what == Event.HANG_UP )    // HANG_UP
18            {
19                freeModems++;
20                System.out.println( "User " + e.who +
21                                    " hangs up at time " + e.time );
22            }
23            else                       // DIAL_IN
24            {
25                System.out.print(  "User " + e.who +
26                                    " dials in at time " + e.time + " " );
27                if( freeModems > 0 )
28                {
29                    freeModems--;
30                    howLong = r.nextPoisson( avgCallLen );
31                    System.out.println(  "and connects for "
32                                       + howLong + " minutes" );
33                    e.time += howLong;
34                    e.what = Event.HANG_UP;
35                    eventSet.add( e );
36                }
37                else
38                    System.out.println( "but gets busy signal" );
39
40                nextCall( freqOfCalls );
41            }
42        }
43    }
```

figure 13.9

The basic simulation routine

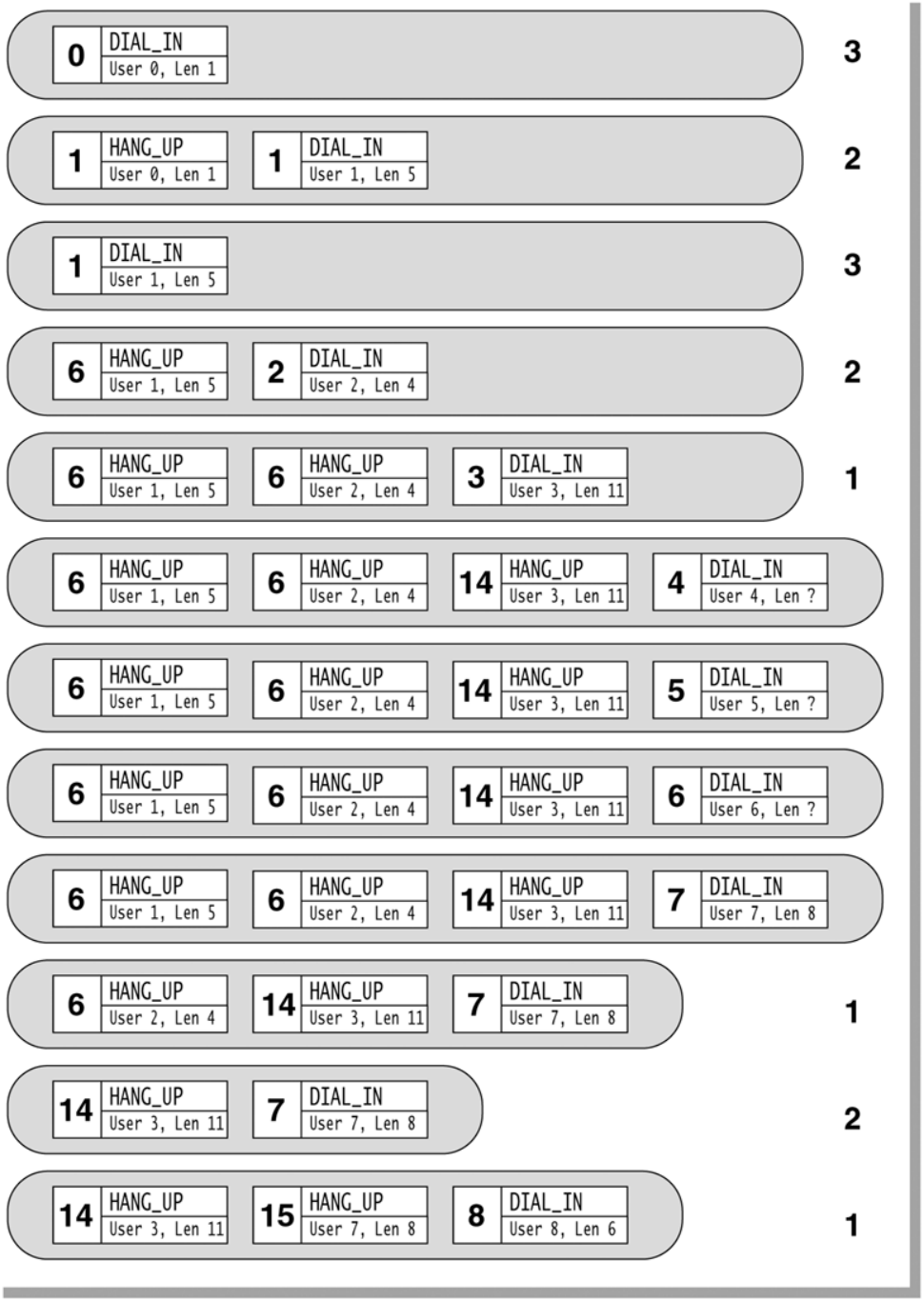


figure 13.10

The priority queue for modem bank simulation after each step

figure 13.11

A simple `main` to test the simulation

```
1       /**
2        * Quickie main for testing purposes.
3        */
4       public static void main( String [ ] args )
5       {
6           ModemSim s = new ModemSim( 3, 5.0, 1 );
7           s.runSim( 20 );
8       }
```

In-Class Activity

- Download the CallSim.java file from CANVAS;
- Run the code;
- Read the code and try to answer the following questions:
  - Is the simulation time-driven or event-driven?
  - How do the events get tracked and processed?
  - How does it keep track of the current time?
  - Can you modify the code to show statistics of the total number of calls received? Timed-out? Average call duration? Average wait time before getting assigned an operator?
    - Hint: add global variables, aggregate their values during each simulation iteration, and then print out their values when the simulation terminates.