Assignment 1: Soccer Ants

Due: The third week of class, with due date and time as per the course information sheet.

Hints, announcements and starter code: On the course web site.

Purpose
To introduce you to queues, binary numbers and random numbers, and also to review inheritance.

Overview
Our program models a farm of ants. These are not normal, everyday ants, however—these ants play soccer! In order to find better soccer-playing ants, we would like to breed them, genetically. We can simulate the breeding using a genetic algorithm. Note that you don’t need any prior knowledge of ants, soccer, genetics or genetic algorithms to complete this assignment, although if you are curious there are many good articles about these topics on the Web.

What you do have to know is that when programming a genetic algorithm, it is common to use binary strings (sequences of bits valued 0 or 1) to model the genes. For this assignment, we’ll implement the gene model, but not the genetic algorithm itself. Our soccer ants’ gene sequences will be represented by binary strings that are 12 bits long, e.g. 1101 1011 0110. The ants have three attributes, which are represented by ranges of bits within the strings. The first 4 bits of the string are for speed, the next 4 bits are for agility, and the last 4 bits are for strength.

To calculate the value of a particular attribute, we take its bit range and interpret it as a binary (base 2) value, converting it into a decimal (base 10) value that we can understand. So a soccer ant with the example string above, 1101 1011 0110, has a speed of 13 units, agility of 11 units and strength of 6 units.

Just to review conversion of binary numbers to their decimal equivalents, note that in order to calculate speed, we look at the four-digit bit range at the beginning of the string (1101) and find that:

- the first bit, in the position with value \(2^3\), is 1
- the second bit, in the position with value \(2^2\), is 1
- the third bit, in the position with value \(2^1\), is 0
- the fourth bit, in the position with value \(2^0\), is 1

So we have \(1(2^3) + 1(2^2) + 0(2^1) + 1(2^0) = 1(8) + 1(4) + 0(2) + 1(1) = 13\).

The Tasks

Task 1: Calculating Speed, Agility and Strength

In our program, we use the Java class SoccerAnt to represent our soccer ants. Each SoccerAnt has a member variable called bits, a boolean array implementing the binary string gene model described above. Inside this array, 0-valued bits are represented as false, and 1-valued bits are represented as true. The attributes and their corresponding bit ranges are as follows:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Bit range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>bits[0..3]</td>
</tr>
<tr>
<td>Agility</td>
<td>bits[4..7]</td>
</tr>
<tr>
<td>Strength</td>
<td>bits[8..11]</td>
</tr>
</tbody>
</table>

The following figure shows what the bits array corresponding to the binary string 1101 1011 0110 would look like (where \(t = \text{true}\), \(f = \text{false}\):

<table>
<thead>
<tr>
<th></th>
<th>t</th>
<th>t</th>
<th>f</th>
<th>t</th>
<th>t</th>
<th>f</th>
<th>t</th>
<th>t</th>
<th>f</th>
<th>t</th>
<th>t</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
Note that for this particular instance, in order to calculate agility, we find that:

- \( \text{bits}[4] \) is true, so the bit in the position with value \( 2^3 \) is 1
- \( \text{bits}[5] \) is false, so the bit in the position with value \( 2^2 \) is 0
- \( \text{bits}[6] \) is true, so the bit in the position with value \( 2^1 \) is 1
- \( \text{bits}[7] \) is true, so the bit in the position with value \( 2^0 \) is 1

So we have \( 1(2^3) + 0(2^2) + 1(2^1) + 1(2^0) = 1(8) + 0(4) + 1(2) + 1(1) = 11 \).

**What to do:** Complete the \text{setSpeed()}, \text{setAgility()} and \text{setStrength()} methods inside class \text{SoccerAnt}. These methods should manipulate the \text{bits} array in order to set the appropriate attribute value (speed, agility or strength). To help you with the programming, we provide the constants \text{ATTRIBUTE_LENGTH}, \text{START_SPEED}, \text{START_AGILITY} and \text{START_STRENGTH}.

**Task 2: Calculating Fitness**

There are four types of soccer-playing ants (implemented as Java subclasses of \text{SoccerAnt}), who play various positions on the soccer field. Strikers, midfielders, defenders and goalkeepers are represented by \text{StrikerAnts}, \text{MidfielderAnts}, \text{DefenderAnts} and \text{GoalkeeperAnts}, respectively.

Eventually, as we breed our ants, we will want to find not only the best ants, but also the best ants given their particular positions. So we need an appropriate fitness evaluation function that can aid our search.

For strikers, we will place more weight on their speed; for midfielders, agility; and for defenders, strength. For goalkeepers, all attributes will be equally weighted. The exact fitness calculations are as follows:

<table>
<thead>
<tr>
<th>Position</th>
<th>Fitness formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Striker</td>
<td>(4(\text{speed}) + \text{agility} + \text{strength})</td>
</tr>
<tr>
<td>Midfielder</td>
<td>(\text{speed} + 4(\text{agility}) + \text{strength})</td>
</tr>
<tr>
<td>Defender</td>
<td>(\text{speed} + \text{agility} + 4(\text{strength}))</td>
</tr>
<tr>
<td>Goalkeeper</td>
<td>(2(\text{speed}) + 2(\text{agility}) + 2(\text{strength}))</td>
</tr>
</tbody>
</table>

**What to do:** Complete the static method \text{getFitness()} inside class \text{Simulation} so that it calculates and returns the appropriate fitness level for a given \text{SoccerAnt}, using the formulae in the above table.

**Task 3: Rigging the Simulation**

The breeding simulation has to start off with an initial population of ants. We could give the ants attribute values using a random number generator, provided to us by the Java Core API through the \text{Random} class. Another possibility is to supply some numbers ourselves, effectively rigging the simulation, so we know what types of ants we are creating. For this purpose, we supply a \text{RiggedIntegerRandom} class, which is a subclass of \text{Random}.

A \text{RiggedIntegerRandom} object reads in a sequence of numbers and cycles through them on calls to \text{nextInt()}. More specifically, when it is first constructed, it reads in a file containing a sequence of numbers (one per line) and enqueues them into a private queue. On each call to \text{nextInt()}, it takes the integer at the head of the queue and returns it, but not before re-enqueuing it so that the numbers can be recycled. (See the comments in the code for more details.)

**What to do:** Complete the \text{RiggedIntegerRandom} constructor so that it reads numbers from the given file and enqueues them into its private queue. Complete the \text{nextInt()} method using the algorithm described above.

**What to Submit**

All your .java files MUST be electronically submitted to your A1 subdirectory or you get a zero. Read the handout Rules for Submission carefully before submitting. You should thoroughly test your code, but you don’t have to hand in any testing. No paper submission is necessary.