CIT 594 – Homework 1

Due – Feb 6, 2017 at 12.00pm

Part 1 – Theory (20 points)

Please do the following problems:

1. Consider an array A of size n >= 2. The array contains numbers from 1 to n-1 (both inclusive) with exactly one number repeated. Describe an O(n) algorithm for finding the integer in A that is repeated. (5 points)
2. What is the order of growth for the following (using the Big-Oh notation)? (5 points each)

```
a. for(int i=1; i<n; i++) {
       for(int j=1; j<5; j++) {
           sum = sum + 1;
       }
   }
b. for(int i=1; i<n; i++) {
       for(int j=1; j<i*i; j++) {
           if(j%i == 0) {
               for(int k=0; k<j; k++) {
                   sum = sum + 1;
               }
           }
       }
   }
```

3. Consider the following two functions and the claim that g(x) is O(f(x)):
   f(x) = 3x^4 + 5x^3 + 17x^2 + 13x + 5; g(x) = x^4
   Prove whether the claim is true or false using the definition of Big-Oh. (5 points)

Part 2 – Programming (80 points)

Organisms: Consider an electronic world consisting of a 10x10 grid. Virtual "organisms" can exist on this grid, with an organism able to occupy a cell on the grid. Organisms have energy that can be gained or lost in a variety of ways. When an organism runs out of energy it dies, and vacates the cell it formerly occupied. An organism can have at most *M* units of energy. An organism may do one of several things during a virtual time cycle:

- Move one cell horizontally or vertically in any direction. The world wraps, so that an organism traveling off the right edge of the grid appears on the left edge, and similarly for the top and bottom edges. A move uses some energy.

- Stay put and do nothing. This move uses a small amount of energy.
- Reproduce. An organism can split in two, placing a replica of itself on an adjacent square. Each resultant organism has slightly less than half of the initial energy of the original organism since reproduction costs some energy.

An illegal move (such as trying to move or reproduce onto an occupied square) results in a "stay put" outcome.

There will be food scattered over the grid. One unit of food corresponds to *u* units of energy. Food reproduces according to the following rules:

- An empty square has a small probability *p* of having a single unit of food "blow in".
- For cells already containing food, but no organisms, every food unit on a nonempty square has a small probability *q* of doubling. This doubling is independent of other food units on the cell. So, a cell with three units of food may have anywhere between three and six units of food on the next cycle. Nevertheless, no cell may have more than *K* units of food at any one time, due to space constraints.
- Cells with organisms never obtain additional food. (The organisms block the light.) In fact, an organism that is on a cell with food, and which has energy no larger than *M*-*u*, will consume a unit of food and add *u* units of energy to its store. If an organism is hungry, it can eat one unit of food per cycle until either the food runs out, or it achieves energy greater than *M*-*u*.

An organism can "see" in the four orthogonal directions. An organism gets information about:

- Whether there is food or not on a neighboring square (but not how much food).
- Whether there is another organism on a neighboring square.
- The amount of remaining food on the organism's current cell.
- The amount of energy currently possessed by the organism.
- Values of the simulator parameters *s*, *v*, *u*, *M*, and *K* (see below), but not *p* or *q*.

Organisms act one-at-a-time from top-left to bottom-right, row by row. We number the top-left cell as (1,1) and the bottom-right cell as *(10,10)*. That means that the state of the virtual world seen by an organism at *(x,y)* reflects the situation in which all organisms in positions *(x',y')* lexicographically less than *(x,y)* have already made their moves, while organisms in positions lexicographically after *(x,y)* have not yet moved. This convention allows all operations to happen without any need for resolving conflicts between organisms (for example trying to move to the same cell). However, it leads to some slightly unintuitive effects:

- An organism that is sensed to the west is usually in its final position for the cycle (can you think of an exception?), while an organism sensed to the east may or may not be in its final position.

- An organism moving east can be sensed from the west, while an organism moving west cannot be sensed from the east. (There's an exception to this observation: what is it?)

You will implement a simulator that implements these rules in Java. The simulator reads in one or more organism brains, places one organism for each such brain randomly on the grid, and lets the organisms behave according to their brains' instructions.

Since there will be many organisms on the grid simultaneously, each will run a separate instance of the Organisms “brain” code. Each instance will have access only to the local environment of the organism. Organisms are placed randomly on the grid, and don't know their coordinates.

The brain can keep a history of local events for the organism if you think that's useful. Organisms cannot identify their neighbors. Neighbors may be of the same species (i.e., have the same programmed brain), or of a different species (i.e., have a different brain).

There are several parameters mentioned in the description. Below is a summary of the parameters, their meaning, and likely values. The first two parameters are not known to the organism, while the remaining parameters are known.

Parameter	**Meaning**	**Likely Values**
p	probability of spontaneous appearance of food	0.001-0.1
q	probability of food doubling	0.002-0.2
s	energy consumed in staying put	1 (other parameters scale)
v	energy consumed in moving or reproducing	2-20
u	energy per unit of food	10-500
M	maximum energy per organism	100-1000
K	maximum food units per cell	10-50

To make things easier (and for the tournament – see EC below), interfaces have been provided. Your code needs to implement these interfaces. They are described below:

1. `OrganismsGameInterface` – "The `OrganismsGameInterface` interface - A Game simulator will implement this interface and will be the starting point of running the game."
2. `Player` – "The `Player` interface - Every Player gets to choose how to move at each turn."
3. `GameConfig` – "This interface specifies the game configuration that is publicly known."
4. `PlayerRoundData` – "The `PlayerRoundData` interface keeps track of how each organism does in a round."
5. In addition to these, the following classes (`Move`, `RandomPlayer`) and interfaces (`Constants`) are also provided, which will be helpful in your implementation.

Requirements: Everyone needs to implement a class each that implement the `OrganismsGameInterface, GameConfig, and PlayerRoundData` interfaces respectively. Everyone needs to implement at least 2 classes that implement the `Player` interface – there should be a `HumanPlayer` (that will ask a user for decisions like moving, staying put, etc. on the command line) and a `ComputerPlayer` (that will use a "strategy" for playing).

Note: Please don't modify any of the given files. This will affect the design and make it harder for the EC.

Part 2 – Extra Credit (20 points)

We will have an in-class tournament (to be run by the TAs after the homework submission date). There are several goals if you decide to attempt the EC:

1. Your organism should be able to survive and replicate in an environment where it is the only kind of organism present. This may not be as easy as it sounds. Overpopulation may lead to consumption of all the food. If *p* is sufficiently small, extinction may ensue. The goal is to achieve the highest long-term stable population.

2. Your organism will be tested in environments containing other organisms. The goal here is primarily to survive, and secondarily to survive in higher numbers than competing organisms. Can your organisms populate the grid faster than their competitors? (Is that even the right strategy given the possibility of everybody going extinct?) How might you program your organisms to "recognize" different organisms based on their state and/or behavior? If you can distinguish members of your species, how might you behave differently to members of another species? Your goal here is not necessarily to obliterate other species, but to maximize the fraction of the population containing your brain.

+5 – If you decide to participate in the tournament, you will get 5 EC points (assuming your code adheres to the specified interfaces and doesn't crash on running). Please indicate yes/no in the readme.txt file and the name of your automated Player class.
+5 – If you finish in the top 50%, you will get an extra +5 points.
+4 – If you finish in the top 25%, you will get an extra +4 points.
+6 – If you finish in the top 5%, you will get an extra +6 points.

For the EC part, you cannot have any help from the TAs/instructor.

Grading Criteria

5% for compilation – If your code compiles, you get full credit. If not, you get a 0.
75% for functionality – Does the code work as required? Does it crash while running? Are there bugs? …
10% for design – Is your code well designed? Does it handle errors well? …
10% for style – Do you have good comments in the code? Are your variables named appropriately? ...

Due to the emergent nature of the Organisms game, it will be difficult to know if there are bugs in the code by playing the game several times. We strongly recommend unit testing your code.

Programming – General Comments

Here are some guidelines with respect to programming style.

Please use Javadoc-style comments.

For things like naming conventions, please see Appendix I (Page A-79) of the Horstmann book. You can also install the Checkstyle plugin (http://eclipse-cs.sourceforge.net/) in Eclipse, which will automatically warn you about style violations.

Submission Instructions

We recommend submitting the theory part electronically also. However, you can turn in a physical copy at the start of class, if you prefer. Please **do not** print out the Java source.

In addition to the theory writeup, you should also submit a text file titled readme.txt. That is, write in plain English, instructions for using your software, explanations for how and why you chose to design your code the way you did. The readme.txt file is also an opportunity for you to get partial credit when certain requirements of the assignment are not met. Think of the readme as a combination of instructions for the user and a chance for you to get partial credit.

Please create a folder called YOUR_PENNKEY. Places all your files inside this – theory writeup, the Java files, the readme.txt file. Zip up this folder. It will thus be called YOUR_PENNKEY.zip. So, e.g., my homework submission would be swapneel.zip. Please submit this zip file via canvas.