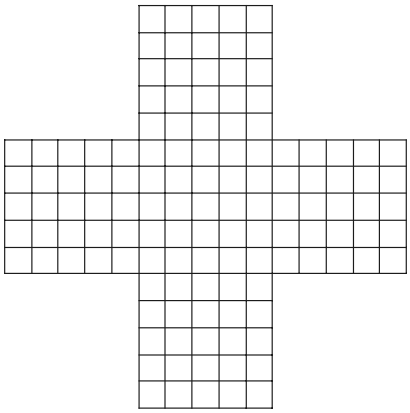
CS 100: Section Assignment 3

(For the week of February 15)

Section assignments are discussed in section and are not submitted for grading. They relate to recent lecture topics and usually to the current Programming Assignment. Prelim questions are based on Section Assignments, Programming Assignments, and Lecture examples.

The discussion of this Section Assignment is special because Prelim 1 is Tuesday, February 16 and because these problems cover Prelim I material. They will be covered in Tuesday sections. For the benefit of students with Wednesday and Thursday sections, there will be a Monday night review session in Olin 155 (7:30-9:00pm) where these problems will be worked. However, all students are welcome at the review session. Moreover, we will publish the solution to these problems on the web Friday February 12. But it is in your interest NOT to look at the answers until you have spent appropriate time thinking about them.

1. We refer to the following graphic as a "T-grid":



A T-grid is made up of five *n*-by-*n* square grids. In the example $n = 5$. Note that a T-grid of size *n* is made up of $3n+1$ horizontal grid lines and $3n+1$ vertical grid lines.

In order to plot a T-grid we identify four design parameters: left (the horizonal coordinate of the left edge), top (the vertical component of the top edge), n (the size), and s the spacing (in pixels) between the grid lines.

Indexing the vertical gridlines 1 to $3n+1$ from left to right, we see that the *k*th vertical gridline is "long" if $k = n+1,\ldots,2n+1$ and is "short" otherwise. Likewise, the *k*th horizontal gridline is "long" if $k = n+1,\ldots,2n+1$ and "short" otherwise.

Write a program that draws a T-grid with n=5, s=20, left = 100, and top = 100. Organize your solution so that it involves a single for-loop that steps from 1 to $3n+1$. The *k*th time through the loop draw the *k*th vertical gridline and the *k*th horizontal gridline. An if-else can be used to determine whether or not the gridlines to be drawn are long or short.

2. The Java program below means well but is incorrect. The intention is to compute rational approximations to π like 22/7. (A rational number is the quotient of two integers.) The integer *q* takes on the values 1,2,…,20 as the loop executes. For each *q* an integer *p* is computed so that *p/q* is as close to π as possible. To compute *p* we (a) multiply *q* by π (e.g., $7\pi = 22.01$) and (b) round the result to the nearest integer (e.g. 22). The mathematics behind the program is correct, but there are mistakes that have to do with type. Correct the program. Do not introduce any additional variables and do not change the type of the given variables.

```
import java.io.*;
public class S3_2{
   public static void main(String args[]){
       TokenReader in = new TokenReader(System.in);
       final int qMax = 20;            // Maximum denominator to try.
       int p,q;                        // p/q is an approximation to pi
       double error;                   // the absolute error in p/q.
       for(q=1;q<=qMax;q=q+1)
       {
          p = Math.round(Math.PI*q);
          error = Math.abs(Math.PI - p/q);
          System.out.println(p + "  " +  q + " " + error);
       }
       in.waitUntilEnter();
   }
}
```

3. Write a Java program that finds the best rational approximation p/q to π with the property that $q <= 10000$. Print the best p and q and the value of $| p/q - \pi |$. For your information, p = 355 and q = 113 renders the smallest error, $| 355/113 - \pi |$ is about 10^{-7} . Of course, there will be a "tie" for the best approximation since (2*355)/(2*113) , (3*355)/(3*113), etc are all rational approximations with equal quality. See if you can organize your program so that it "chooses" 355/113.

Proceed by modifying the "corrected" version of the program given above. Each pass through the loop you must check to see if p/q is the "best" quotient found so far. If so, you must "remember" those p and q and the associated error. For that purpose, use variables pBest, qBest, and smallestSoFar . Every time your program discovers a better rational approximation it should update these variables accordingly.

4. In the game of "Up and Down" you start with an integer greater than one and generate a sequence by repeating this process until the integer one is reached:

If the current integer is even, then get the next integer by dividing by two.
If the current integer is not even, then get the next integer by multiplying by 3 and adding 1.

Here is what happens when 17 is picked as the starting integer::

52, 26, 13, 40, 20, 10, 5, 16, 8, 4 ,2 , 1 (Score = 12)

Here is what happens when 100 is picked as the starting integer:

50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2 , 1 (Score = 25)

As you can see, the "score" is the number of turns it takes to reach unity. (Interestingly, it can be shown that the sequence will always reach unity.)

Write a program that solicits a starting integer and prints the score. Use a while loop and type long variables. Although the up-and-down sequence is guaranteed to reach unity, organize your program so that the while loop terminates after (say) 100 turns. If it requires more than 100 turns to reach unity then the program should not print the score but the message "More than 100 turns required to reach unity."