COS125 - Precept 4 (Loops)

1 Tracing Loops

For each value of n, write down the value of variable `counter` at the end of each code snippet below.

Code	$n = 2$	$n = 8$	$n = 128$

`int counter = 0;`
`for (int i = 0; i < n; i++)`
`  counter++;`
	2	8	128

`for (int i = 0, counter = 1; i < n; i++)` `  counter *= 2;`			

`int counter = 0;`
`while (counter < n)`
`  counter++;`			

`int counter = 0;`
`for (int i = 1; i <= n; i *= 2)`
`  counter++;`			

`for (int i = 0, counter = 0; i < n; i++)`
`  for (int j = 0; j < n; j++)`
`  counter++;`			

`int counter = 0;`
`for (int i = 1; i <= n; i *= 2)`
`  for (int j = 0; j < n; j++)`
`  counter++;`			

2 Reverse Number

Write a program that takes in an integer of any size as a command line argument and prints out the number in reverse using integer operations only (i.e., do not reverse the number with string operations). You can assume the integer passed in by the user does not have a leading zero (e.g., 0123 is an invalid argument).

Example:

```
~/precept4/$ javac-introcs ReverseNum.java
~/precept4/$ java-introcs ReverseNum 123
321
```

3 Computing π

Calculating π can be done using an approximation with the following steps and observations:

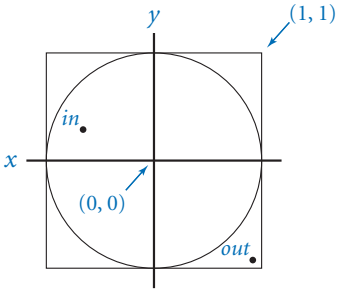
1) Superimpose a circle of radius 1 centered at the origin (0,0) on top of a 2 x 2 square also centered at the origin.

2) The area of this circle is simply π because $A = \pi r^2 = \pi(1)^2 = \pi$. The area of this square is equal to 4.

3) The probability that a random point within the square also falls within the circle will be approximately equal to the ratio between the area of the circle and the area of the square ($\pi/4$).

Optional:

Write a program `MonteCarloPi.java` that computes an approximation to $\pi \approx 3.1416\ldots$ with the following Monte Carlo simulation: reading a number of iterations n from the command line, initialize a variable `hits = 0` and repeat the following n times:



1. Sample a (uniformly) random point (x, y) from the square $[-1, 1] \times [-1, 1]$.
2. Check if the point lies inside the unit circle: if $x^2 + y^2 \leq 1$, increment `hits` by 1.

Finally, divide `hits` by n and then multiply by 4 (see Observation 3 above) and print the result. (*Hint: you may find a particular lecture slide useful.*)

4 Dice Battle Simulation

Download `precept4.zip` from the precepts webpage. Unzip and open the project folder.

In many tabletop board games, you may run into dice with a variety of sides. For instance, 6, 8, 10, and 20 (referred to as d6, d8, d10, and d20) are common n-sided dice. In some games, you roll against an opponent and whoever has the higher number wins. Before challenging a difficult opponent, you may want to know your odds of winning, especially if you are rolling a smaller sided die than they are.

A Monte Carlo simulation is a program that helps us simulate random events repeatedly, enough to approximate a statistic. We can create a Monte Carlo simulation to calculate the odds of winning a dice roll against an opponent.

Write a program `DiceBattleSim.java` that takes in three integers, the first is the number of sides of your die, the second is the number of sides on your opponent's die, and the third is the number of times to roll. Your program should then simulate rolling these dice against each other the number of times specified in the third argument. Calculate your approximate odds of winning

by printing the ratio of how many times you won divided by the total number of rolls (wins + losses).

Once you get this version working, add in these variations to your simulation:

1) If you roll a 1, you immediately lose no matter what.

2) If you roll a 20 (assuming you roll a d20), you win no matter what.

3) Ties require a reroll.

Try to handwrite the math for calculating dice roll odds and see if your math matches your simulated odds.

5 Greatest Common Divisor

Euclid's Algorithm

Create a program named `Euclid.java` that implements Euclid's algorithm for finding the greatest common divisor (GCD) between two numbers. Your algorithm should take two positive `long` command-line arguments.

Euclid's algorithm proceeds as follows: set r_1 to be the larger and r_2 the smaller between a pair (a, b) of positive integers, and generate a sequence by setting r_n to be the remainder of the division of r_{n-2} by r_{n-1} (recall that `%` is the Java operation for the remainder). The algorithm terminates when $r_n = 0$, and the GCD is r_{n-1}.

For instance, if $r_1 = 20$ and $r_2 = 5$, then $r_3 = 20 \ \% \ 5 = 0$. Since $r_3 = 0$, then the GCD of 20 and 5 must be stored in r_2 which is 5.

6 Smallest Bill Denomination[1]

Complete the program `SmallestBill.java` below, which takes in at least 1 argument. This argument is an integer, *num*, of at least 0 that represents how many more arguments are passed in. The next *num* arguments each represent a dollar cost (between \$0 and \$50) for an item. For each item cost in the arguments, this program should print the smallest dollar bill denomination (\$1, \$5, \$10, \$20, \$50) that can cover its entire cost.

Example:

```
~/precept4/$ javac-introcs SmallestBill.java
~/precept4/$ java-introcs SmallestBill 2 20 25
$20
$50
```

Fill out the code on the next page.

[1] Mid-term exam practice

SmallestBill.java:

```
public class SmallestBill {
    /* Takes in at least two integer arguments:
     a number, n, representing how many arguments are left
     and then the next n args each representing the price (< $50) of an item.
     Prints out the smallest denomination that will pay for each good. */
    public static void main(String[] args){

        // Your code here









    }
}
```