CS 103 – Going Broke

1 Introduction

In this exercise you will program a simulation of a simple coin-flipping game to find the average number of flips before one of the players goes broke. The scenario is that of three bored students with a large number of quarters (presumably for the laundry machine). To ease their boredom they agree to play a game where each turn consists of all three students flipping one of their coins, examining the resulting combination of heads and tails. If all three coins match (heads or tails), each player takes their coin back. Otherwise, whoever threw the one coin that does NOT match the two others gets all three coins, and the game repeats. Assuming, all players start with n coins, what is the average number of flips before one (i.e. the first) player goes broke?[1]

This programming assignment should be performed **INDIVIDUALLY**. You may re-use portions of code from previous examples and labs provided in this class, but any other copying of sections of code is **prohibited**. We will use software tools that check all students' code and known Internet sources to find hidden similarities.

2 What you will learn

After completing this programming assignment you will:

1. Write a non-trivial C program from scratch by combining knowledge from previous examples and modifying them appropriately
2. Select appropriate control structures including for loops, while loops, conditionals / selection mechanisms (i.e. if statements) to solve a problem
3. Use the random number generation capabilities of the standard C library.
4. Combine and refine the above skills to develop a computer simulation of a non-trivial random process.
5. Assess the time complexity of your algorithm

3 Background Information and Notes

We will execute a large number of game simulations to arrive at an average number of turns for each game (this process of simulating a large number of experiments to find the probabilistic answer is known as Monte Carlo simulation due to its resemblance to how casinos bank on the fact that games of chance will yield on average what probabilistic mathematical formulas predict). This can be easily accomplished by writing a kernel of code for a single game simulation then repeating it for some large number of times (say 100,000). By tracking how many turns each game required over the course of 100,000 game simulations and then dividing by 100,000 we can arrive at

[1] Taken from Nahin, Digital Dice, Princeton University Press, 2008 and originally published by G.W. Petrie in *American Mathematical Monthly*, August-September 1941 issue.

the average number of flips per game. <u>Note: While every game has an integral number of turns, the average may not be an integer</u>.

One of the primary reasons that it may be easier to simulate this problem rather than attempting to find an analytic solution (besides the fact that you probably have not had a probability class) is that to simulate a coin flip we can just use the random number generation capabilities of the standard C library (or any another programming environment) through the `rand()` function. Most random number generators will produce a number that is **uniformly** distributed over a certain range. A uniform distribution means that theoretically every number over the range has equal probability of being produced. `rand()` returns an integer number in the range 0 to `RAND_MAX` (which is a constant defined in `cstdlib` which is also where `rand()` is declared/prototyped as well). You will need to think about how to convert that to an appropriate value to represent the outcome of a single coin-toss.

One issue with `rand()` and most computers generating random numbers is that computers do only what they are told and so it is sometimes hard to be random. Most computers generate pseudo-random numbers (sequences that look random but are not necessarily). And in fact, because of this, every time you start your program you will be the SAME random sequence. To get a DIFFERENT sequence for each program run, we need to "seed" the random number generator with a different starting value. You can do this with the `srand(int seed)` function in `cstdlib`. During testing we often provide the same seed to make sure our results are similar as we update the code. When we are satisfied our code works we can provide a unique seed each run (the easiest way to do this is to pass srand() the current time which should be different for each run). Thus, as you develop your program place the following line of code as your first statement after declaring your variables.

```
  srand(0);
```

Then when you think your code is working, change the line of code to read:

```
  srand( time(0) );
```

You will need to include `<ctime>` in addition to `<cstdlib>` to use this functionality.

4 Requirements

Your program shall meet the following requirements for features and approach:

1. Prompt the user to enter **a.)** the initial number of coins (just one number…all 3 players will start with that number), **b.)** the fairness (probability of heads) of the coin as a decimal between [0,1), and **c.)** the number of game simulations to perform
2. Receive input from the user for the above three values, **<u>in the order</u>**:
   a.) number of initial coins, b.) coin fairness, c.) number of game simulations
3. **Use a 3-element array to maintain the current number of coins for player 0, 1, and 2.**
4. Use nested loops to perform the specified number of game simulations where each game is a sequence of turns/rounds.
5. Maintain a running sum of all the turns required for all the simulated games thus far.
6. Output the average number of turns required over all the simulations. You **MUST use the format** shown below for printing out your result.

```
Average: 1.33
```

5 Prelab

Download the **`readme.txt`** skeleton file provided at: http://bits.usc.edu/files/cs103/coins/readme.txt and enter your answers to the prelab questions below. Your answers can just be plain English summaries of your approach to the questions below. Do this **BEFORE** you begin to program or write code.

1. How will you generate a coin-toss outcome for each player? (i.e. map the random number returned by rand() to a Heads/Tails outcome)
2. Based on the coin-toss outcome for each player, what cases do you need to check for to decide how to do the coin bookkeeping? What logical connectors do you use?
3. Is the number of iterations/turns that a single game requires known **a priori** (which simply means "beforehand" but is used quite often in computer science speak). What kind of loop should then be used and what is its terminating condition?
4. What data needs to be maintained for the total simulation (i.e. across several games)? What type?
5. Is the number of simulations known before entering the loop? What kind of loop should then be used?
6. Also, consider the case that somehow the players all use unfair (possibly weighted) coins that come up as heads with probability, p. We will start by simulating fair coins where p=0.5. Assuming that for fair coins (p=0.5) we calculate *x* number of flips per game. How will the number of flips per game change if p = 0.1? Will it stay the same or go up or down and how dramatically? Think about this scenario and write a sentence of two explaining your prediction and *your reasoning* before starting to code your solution.

6 Procedure

Perform the following.

1. Using your answers to the problems above implement the Monte Carlo simulation as a C/C++ program, in a file named `broke.cpp`.
2. Checkpoints: If you are new to programming or having trouble getting started try to write the code in stages and be sure each stage is working. See the debugging strategies below for some testing approaches as your write each stage:
    a. **Stage 1**: Start by writing a program to prompt the user for the necessary input values [in the order specified in the Requirements section] and then read them in from the keyboard into appropriate variables…Check your work by displaying them back to the user.
    b. **Stage 2**: Write the code that simulates a single turn where 3 coin tosses are simulated for the 3 players and then updates the number of coins each player will have for the next turn. This is a simulation of a single turn.
    c. **Stage 3**: Wrap the code from the previous step in some kind of loop that will continue making coin tosses / turns until one player has no more coins remaining. This is a simulation of a single game.
    d. **Stage 4**: Wrap the code from the previous section in some kind of loop to execute the Monte Carlo simulation or the specified number of trials/games and computes the average number of turns per game.
3. Comment your code as you write your program documenting what abstract operation a certain for, while, or if statement is performing as well as other decisions you make along the way that feel particular to your approach.
4. Compile and run your program:
```
$ make broke
```
5. The following table is a list of results for a few input values. Use it to judge the whether your program seems to be working. If it is not, you may use a debugger such as '`gdb`' (i.e. `$ gdb broke`).

Init. # of Coins	Prob. of Coin	Game Simulations	Result (Avg. # of turns)
1	0.5	100,000	1.33
3	0.5	100,000	5.13 – 5.14
5	0.5	100,000	12.7 - 12.8
4	0.7	100,000	10.1-10.2

6. Next, experiment with number of game simulations used for the case of fair coins (p=0.5) and 3 initial coins for each player. Run your program several times varying the number of trials/simulations for n=5, 10, 50, 100, 1,000, 10,000, and 100,000. Create a neatly formatted table showing the number of trials and the resulting average number of turns. This experiment is important so that we can understand how the results change based on number of trials. Why run more

trials and wait longer if an accurate result can be found with fewer trials? Use your table to suggest the point of "diminishing returns" (indicate a number of simulations to perform) where performing more games simulations does not improve the accuracy.

7. Finally, perform an experiment using 3 initial coins for each player and running 10,000 experiments for each run, but now letting the fairness of the coin (i.e. p) range from 0.1 to 0.9 by increments of 0.1 (i.e. 0.1, 0.2, 0.3, …, 0.9]). Record the answers and create another table listing the probability and the average number of turns.

7 Debugging Strategy

When you are learning to program, learning to debug is just as important as learning how to write the code. So to that end, here is a strategy I've used for this PA. The key is to place print/cout statements at key locations in your code so you can see where you are and when. Then print out variable values at those times.

So here's my strategy:

1. After stage 1, just trying printing out the variable's you've received as input and the initial value of the coins.
```
input: 3 0.5 10

output:
starting 3
fairness 0.5
trials 10
coins 3 3 3
```

2. After completing the stage 2 portion of your program you could add outputs that indicate a single turn is working by printing the outcome of each coin flip and the resulting number of coins each player has:
```
input: 3 0.5 10

output:
starting 3
fairness 0.5
trials 10
coins 3 3 3
heads? true true false
coins 2 2 5
```

Note: To see the words 'true' and 'false' printed for Boolean variables you have to tell cout to print Booleans as words by setting the 'boolalpha' flag as in:

```
cout << boolalpha << my_bool_variable;
```

Otherwise you will just see 0 (for false) and 1 (for true). Either way is fine since this is just temporary for debugging purposes.

3. After stage 3 your program could show the output of an entire game.

```
input: 3 0.5 10
output:
starting 3
fairness 0.5
trials 10
coins 3 3 3
heads? true true false
coins 2 2 5
heads? false false false
coins 2 2 5
heads? false true false
coins 1 4 4
heads? false false true
coins 0 3 5
flips 4
(or some other sequence of flips)
```

4. After stage 4 you can run the game multiple times and print out the average number of turns that a game will last.

```
input: 3 0.5 10
output:
5.2
(or some other number)
```

Important: You must remove all of these "debugging" print statements and just show the final answer in the version of code that you submit.

8 Review

1. Use your experiment data for the number of trials to suggest the point of "diminishing returns" where the extra simulations do not yield an improvement in accuracy worth the extra simulation time.

2. Review the experiment data as p varies for the 3-coin case. Do these results match your expectations of the behavior of the average number of flips as p approaches 0 or 1?

3. When we examined the effects of varying p, why did we start at 0.1 and end at 0.9 rather than starting at 0.0 and ending at 1.0?

9 Lab Report

Include the following items in your **`readme.txt`** that you downloaded earlier for your prelab questions.

1. Include your name, lecture section/time, and USC ID at the top
2. Then type your answers to the prelab.
3. Include a neatly formatted (but just in plain text) table of the number of trials and the accuracy of the result.
4. Your neatly formatted table of p and the average number of flips
5. Answers and explanations to the answers posed in the Review section.

10 Submission instructions

Submit your code and readme file on the class website.