Lab 1: Simulation and OOP

Initial Instructions

Create a Java project called Lab1 (assuming you use Eclipse) and unzip lab1.zip into the src folder. Refresh your project in Eclipse.

Part A: Simulation

Suppose a sports league has 12 teams. During the season each pair of teams plays exactly once (i.e., this is not an elimination tournament). Every game results in a win or loss. A win counts as 1 point, and a loss is 0 points. We will assume that the teams are all perfectly equal in skill, so each game reduces to a fair coin flip. At the end of the season, the team(s) with the highest score are considered the best. On average, what will be the best score for the best team(s) with the highest score?

To solve this problem we will perform a (Monte Carlo) simulation. Your program should take 2 command-line arguments (i.e., it should get them through args): \( n \), the number of teams in the league (we had \( n = 12 \) above), and \( m \), the number of simulation rounds. You will then simulate a season \( m \) times. On each simulation, compute what the score of the highest scoring team is. Then take an average of this value over all seasons. Output the average number of wins on the console. Run your simulation for \( m = 100000 \) seasons with \( n = 12 \) and then again with \( n = 20 \). Round your answer to the nearest hundredth and let me know when you think you have it correct. What is the runtime of your implementation in terms of \( m \) and \( n \)?

Implementation suggestions:

1. If you haven’t worked with command-line arguments before, make your initial implementation take \( m \) and \( n \) from args, convert them to ints, and print them to the console. Then test that you are able to get \( m, n \) correctly.

2. Break your program into a few smaller functions that each have a simple task. It may help to spend a few moments planning this out ahead of time.

3. You can make a static java.util.Random object for all of your functions to use. It has a nextBoolean method that flips a coin.

4. Let \( m = 1 \) and \( n = 5 \) and print out the return values of all of your functions to test them (not the greatest test, but you can certainly eyeball the output to make sure nothing crazy happened). If you need to turn an array into a String, use the function Arrays.toString.
Part B: Pile Game

In the Pile game there is a number of piles that each have stones. The 2 players of the game alternate by making moves as follows: a player must choose a pile with some stones in it, and then must remove a positive number of stones from that pile. A player loses if it is their turn and no stones remain. The class PileGame will currently allow 2 human players to play the Pile game against each other when it is run. Try playing the game to get a feel for it.

Your job is to make some computer AIs for the Pile game that implement the interface PileStrategy. Here are the AIs you are required to code up:

1. RemoveFirst: This strategy will always remove all stones from the first (lowest-index) non-empty pile.

2. PureRandom: This strategy randomly selects a non-empty pile (with all non-empty piles equally likely) and then randomly selects a positive number of stones to remove that is at most the size of the pile (all valid positive amounts equally likely).

3. (⭐⭐⭐) PerfectPlay: This strategy will play perfectly. In other words, if its position is winnable (in any number of moves) it will always make the correct set of moves to force a win.

To add your strategy to the game, add an object of your strategy’s type to the array strats in the main function of PileGame.

What You Submit

If you are teaming up with someone on this lab, create a text file called Collaborators.txt and put all team member names in the file. Put all of your source code (Parts A and B) and Collaborators.txt into a zip file and upload it on NYU Classes. Make sure to upload the zip file to everyone’s classes account (e.g., each of the 2 team members should upload the same zip file).