BEE 4750 Lab 2 Solution

Setup

The following code should go at the top of most Julia scripts; it will load the local package environment and install any needed packages. You will see this often and shouldn't need to touch it.

```
import Pkg
Pkg.activate(".")
Pkg.instantiate()
```

```
using Random # random number generation
using Distributions # probability distributions and interface
using Statistics # basic statistical functions, including mean
using Plots # plotting
```

Exercises (10 points)

```
Random.seed!(1)
```

```
TaskLocalRNG()
```

Problem 1 (5 points)

We want to know the probability of getting at least an 11 from rolling three fair, six-sided dice (this is actually an old Italian game called *passadieci*, which was analyzed by Galileo as one of the first examples of a rigorous study of probability).

Problem 1.1 (1 point)

Write a function called `passadieci()` to simulate this game, which will take as an input the number of realizations and output a vector of the sum of the three dice rolls for each realization.

Solution:

```
# n is the number of realizations
function passadieci(n)
    outcomes = zeros(n) # initialize storage for realizations
    for i = 1:n
        dice_dist = DiscreteUniform(1, 6) # create dice roll distribution
        outcomes[i] = sum(rand(dice_dist, 3)) # add three draws from the distribution
    end
    return outcomes
end
```

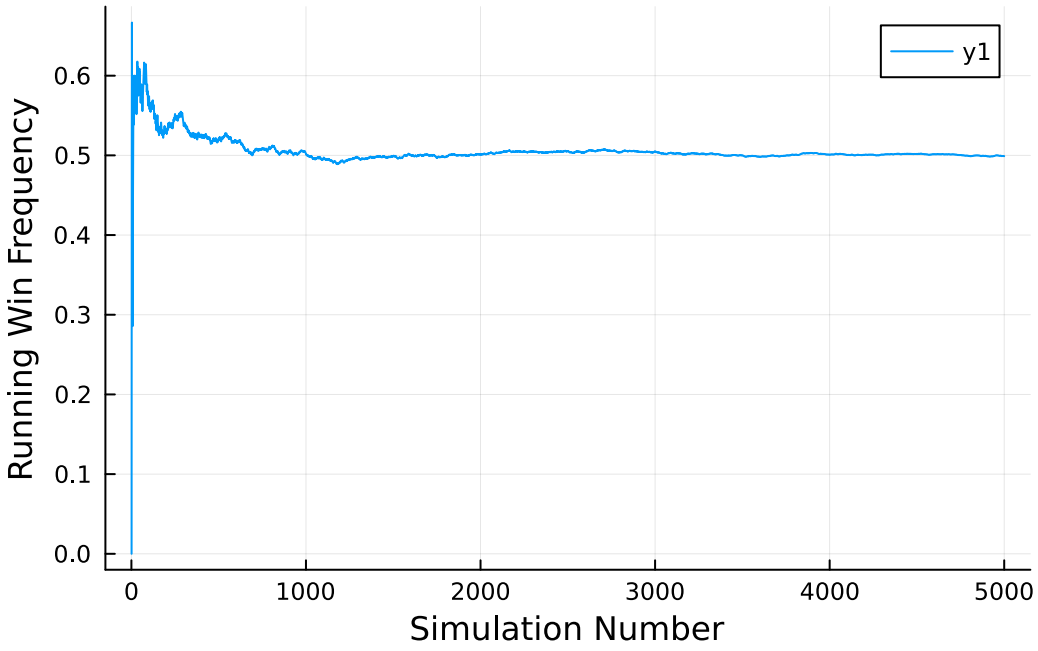
```
passadieci (generic function with 1 method)
```

Problem 1.2 (2 points)

Generate 5,000 simulations of the game using your `passadieci()` function. Plot how the computed probability of winning the game changes as the number of simulations increases (you can do this by computing the frequency of wins for each additional simulation).

Solution:

```
passadieci_sims = passadieci(5000) # simulation
# compute running number of wins
wins = passadieci_sims .>= 11 # translate dice rolls into wins
win_freq = zeros(length(passadieci_sims))
# compute average number of wins by looping over and adding the impact of the next game
for (i, next_win) in pairs(wins)
    if i == 1
        win_freq[i] == next_win
    else
        win_freq[i] = (win_freq[i-1] * (i-1) + next_win) / i
    end
end
# make plot
plot(win_freq, xlabel="Simulation Number", ylabel="Running Win Frequency")
```



Problem 1.3 (2 point)

Based on your plot from Problem 1.2, how many simulations were needed for the win probability estimate to converge? What did you notice from your plot about the estimates prior to convergence?

Solution:

The probability sememed to converge after around 1,500–2,000 simulations. We can see that the estimates are relatively noisy prior to this point, even if the overall trend is to move towards the eventual average.

Problem 2 (5 points)

The Showcase is the final round of every episode of The Price is Right, matching the two big winners from the episode. Each contestant is shown a "showcase" of prizes, which are usually some combination of a trip, a motor vehicle, some furniture, and maybe some other stuff. They then each have to make a bid on the retail price of the showcase. The rules are:

- an overbid is an automatic loss;
- the contest who gets closest to the retail price wins their showcase;
- if a contestant gets within $250 of the retail price and is closer than their opponent, they win both showcases.

Your goal is to find a wager which maximizes your expected winnings, which we may as well call utility, based on your assessment of the probability of your showcase retail price.

We'll assume that the distribution of all showcases offered by the show is given as truncated normal distribution, which means a normal distribution which has an upper and/or lower bound. `Distributions.jl` makes it easy to specify truncations on any distribution, not just normal distributions. For example, we'll use this distribution for the showcase values:

```
showcase_dist = truncated(Normal(31000, 4500), lower=5000, upper=42000)
```

```
Truncated(Normal{Float64}( =31000.0,  =4500.0); lower=5000.0, upper=42000.0)
```

Problem 2.1 (3 points)

Write a function `showcase()` which takes in a bid value and uses Monte Carlo simulation to compute the expected value of the winnings. Make the following assumptions about your expected winnings if you don't overbid:

- If you win both showcases, the value is the double of the single showcase value.
- If you did not win both showcases but bid under the showcase value, the probability of being outbid increases linearly as the distance between your bid and the value increases (in other words, if you bid the exact value, you win with probability 1, and if you bid \$0, you win with probability 0).

How did you decide how many samples to use within the function?

```
# the syntax (...; param=val) is used for optional
# arguments, which are given a default. these arguments must be
# named in subsequent calls if they are changed, as we will see below.
function showcase(bid; nsamples = 500000)
    samples = rand(showcase_dist, nsamples)
    winnings = 0 # running total of winnings over the simulations
    for (i, value) in pairs(samples)
        if 0 < (value - bid) < 250
            winnings += 2 * value
        elseif bid < value
            p_win = bid / value
            winnings += value * p_win
        end
    end
    # return expected winnings
    return winnings / nsamples
end
```

```
showcase (generic function with 1 method)
```

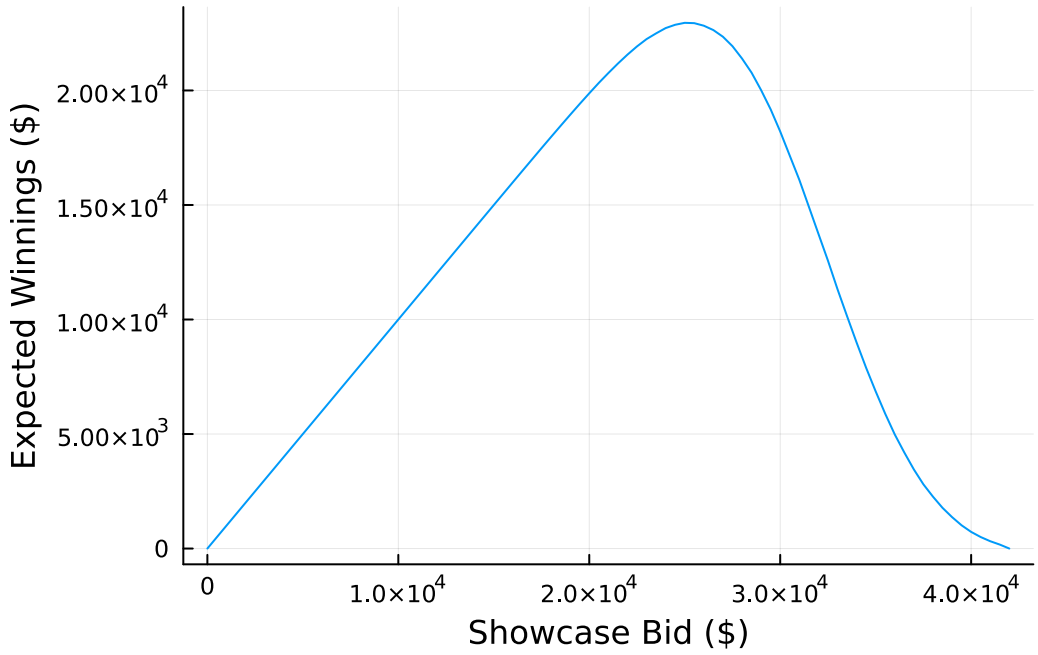
One could justify the number of samples by looking at the plot of the Monte Carlo estimate as additional samples are created (for example, analogously to Problem 1.2). However, this should not be done arbitrarily.

Problem 2.2 (2 points)

Plot the expected winnings for bids ranging from \$0 to \$42,000. What do you notice?

Solution:

```
bids = 0:500:42000
expected_winnings = showcase.(bids)
plot(bids, expected_winnings, xlabel="Showcase Bid (\$)", ylabel="Expected Winnings (\$)", la
```



The bid is maximized around \$25,000, which we can see with the following code:

```
# get index 2 from findmax, which returns both the value and the index of the maximum
best_idx = findmax(expected_winnings)[2]
bids[best_idx]
```

25000

Finding this value explicitly was not required, but at least some analysis of the plot would be useful! More than that starts to increase the risk of an overbid, while less than that and we open up space for our opponent to outbid us. Due to the asymmetry of the problem, the dropoff from the "best" bid is quite rapid; this is similar to when we have a strong constraint on an environmental system (such as a regulatory limit) and there can be a large penalty for exceeding it.

Also, this is a point where if we had used too few samples in Problem 2.2, we might have noticed that our plot would be less smooth than expected due to the lack of convergence. This could distort our analysis of the resulting plot/simulations.

References

Put any consulted sources here, including classmates you worked with/who helped you. :::