Lab 2 Solutions

> Due Date
>
> Wednesday, 9/25/24, 9:00pm

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
```

Exercise (3 Points)

First, let's write a function which compares our bid to (random) showcase value and returns the reward. The only slight complication is calculating the probability of winning if we don't automatically win both showcases and don't overbid. Our assumption is that this is linear, with a win probability of zero when we bid \$0 and a win probability of one if we bid exactly. So, using the equation of a line between these two points, if our bid is b and the true value is v, the probability becomes

$$\mathbb{P}(v-b) = 1 - \frac{v-b}{v}$$

and the expected value of our winnings in this case is $\mathbb{P}(v-b) \times v = b$.

```
function showcase_play(value; bid=35_000)                                    ①
    if bid > value # overbird, win nothing
        return 0
    elseif value - bid < 250 # win both showcases
        return 2 * value
    else
        win_prob = (1 + (bid - value) / value)
        return bid
    end
end
```

① The semi-colon and assigned value makes `bid` an optional parameter: if we don't explicitly pass it, it will take on that assumed value, but we can change it by calling `showcase_play(value; bid=x)`.

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

Now we can conduct our Monte Carlo experiment. Let's write another function which takes in a vector of showcase values and computes the running average of winnings.

```
function showcase_mc(values)
    exp_winnings = zeros(length(values))                                     ①
    for (i, value) in pairs(values)                                          ②
        if i == 1
            exp_winnings[i] = showcase_play(value)
        else
            exp_winnings[i] = ((i - 1) * exp_winnings[i - 1] +
 ↪  showcase_play(value)) / i                                                ③
        end
    end
    return exp_winnings
end
```

① This initializes the storage for a vector which will store the running average for every iteration.

② `pairs(v)` returns an iterator over tuples containing both the index and the value of each element of `v`. This is safer than `for i = 1:length(v)` for a larger variety of data structures and avoids having to explicitly look up `value = v[i]`. If we didn't need the index, we could have used `for value in values`, but we want to be to write to the right index in `exp_winnings`.

③ We could just compute the average each time over the entire chunk of the vector, but this is a little faster for cases where the evaluation is more expensive as we avoid re-computing.

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

Now if we draw 20,000 samples (this is large for illustrative purposes), we can compute how the Monte Carlo estimates change (visualized in Figure 1).

```
Random.seed!(1)                                                    ①
showcase_dist = truncated(Normal(31000, 4500), lower=5000, upper=42000)
showcase_samples = rand(showcase_dist, 20_000)

winnings_mc = showcase_mc(showcase_samples)
plot(winnings_mc, xlabel="Monte Carlo Iteration", ylabel="Expected Winnings
 ↪  (\$)", label=false)
```

① Setting a seed ensures we'll get the same samples regardless of when we re-run this notebook.

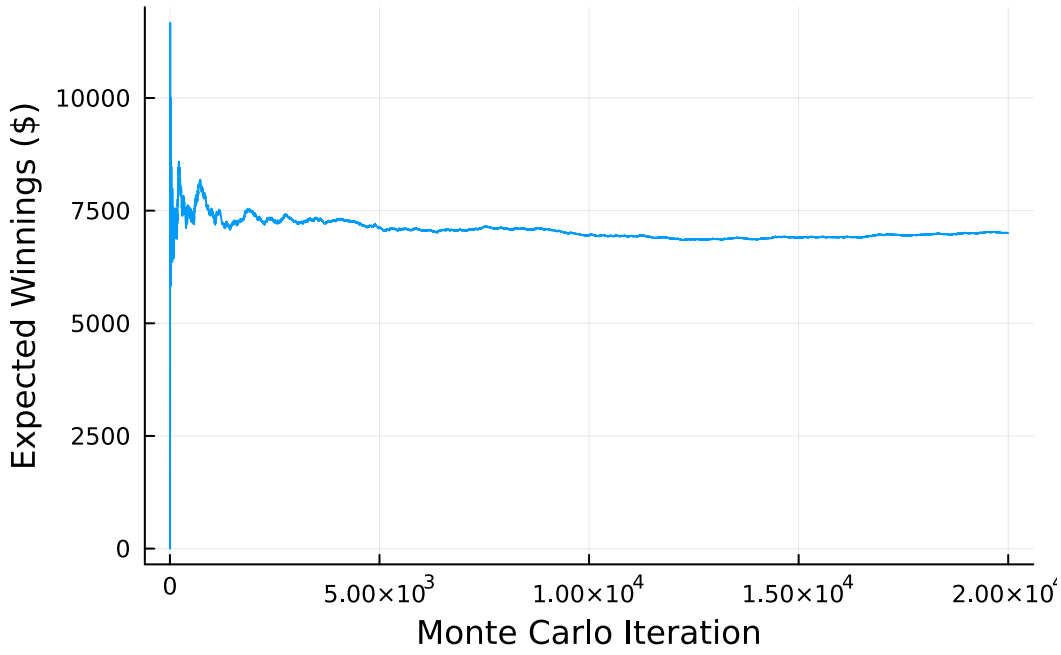

Figure 1: Monte Carlo estimates of expected winnings for a $35,000 bid over 20,000 showcase samples.

Based on this, the estimated value is $6999.

How could we decide how many samples to use? Later, we'll see in class how we can compute a more formal analysis of the error in the Monte Carlo estimate, but for the purposes of this lab, we would like to ensure that the estimate has stabilized. If we had only used the first 1,000 samples, we would see Figure 2.

```
plot(winnings_mc[1:1000], xlabel="Monte Carlo Iteration", ylabel="Expected
 ↪  Winnings (\$)", label=false)
```

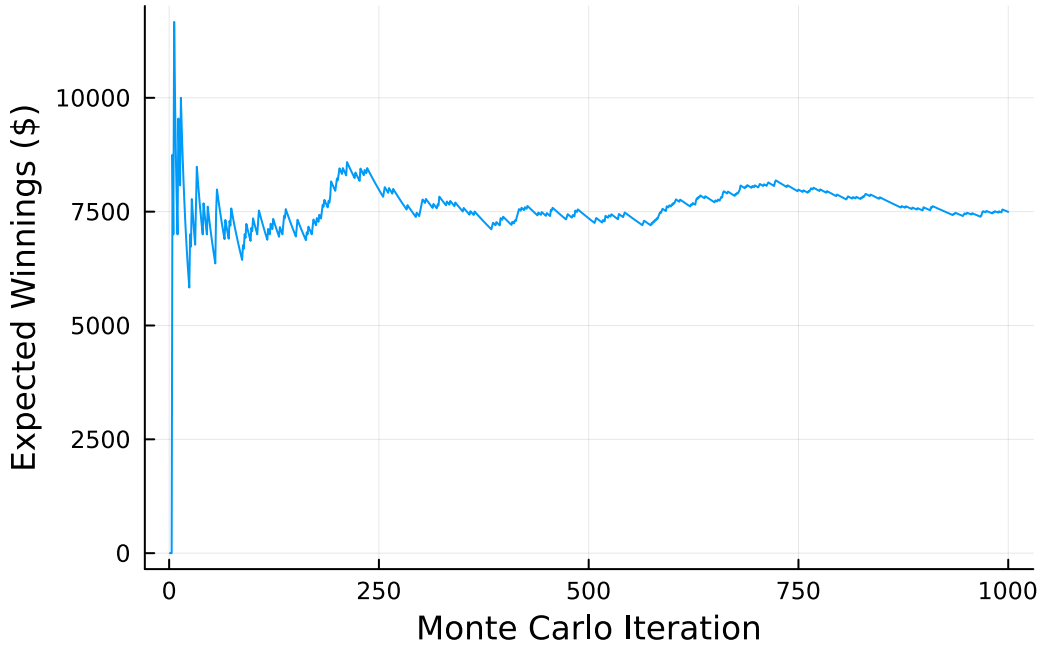

Figure 2: Monte Carlo estimates of expected winnings for a $35,000 bid over 1,000 showcase samples.

We can see that there is still some variability as of the last iteration, so we might want to use more samples.[1] If we use 5,000 samples, we can see the results in Figure 3.

```
plot(winnings_mc[1:5000], xlabel="Monte Carlo Iteration", ylabel="Expected
 ↪  Winnings (\$)", label=false)
```

[1]Note that this variability might be ok depending on the outcomes of the error analysis and our desire for precision, but more on that soon.

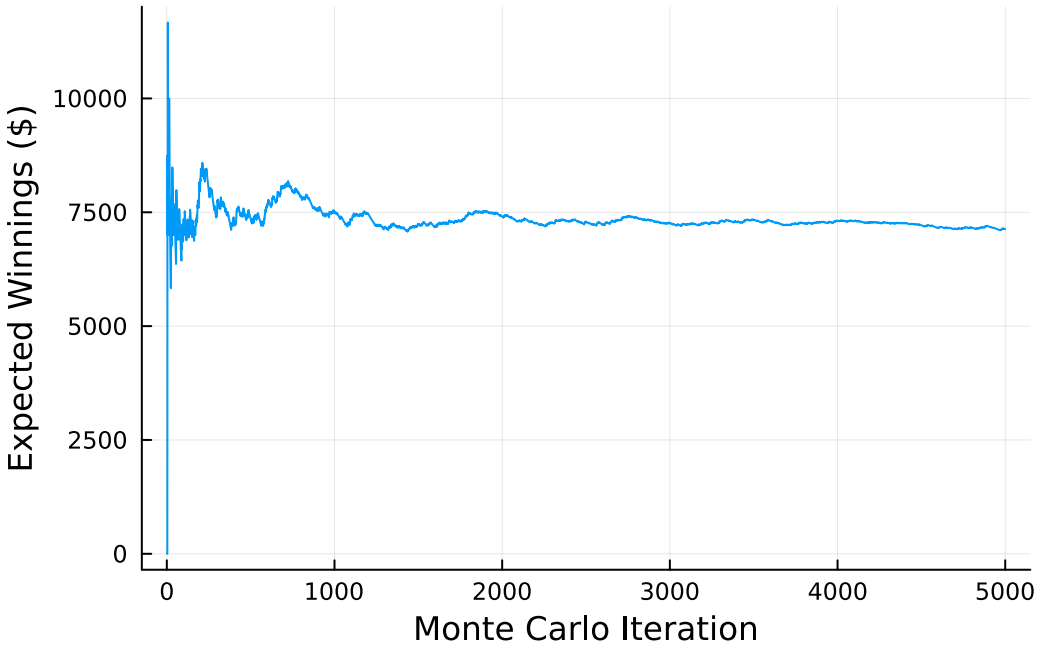

Figure 3: Monte Carlo estimates of expected winnings for a $35,000 bid over 5,000 showcase samples.

This looks much better! So we could have used 5,000, but it will turn out that more will always be "safer".

References

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