Stat 110 Strategic Practice & Homework 3: Gambler's Ruin

Problem

A gambler repeatedly plays a game where in each round, he wins a dollar with probability 1/3 and loses a dollar with probability 2/3. His strategy is "quit when he is ahead by $2," though some suspect he is a gambling addict anyway. Suppose that he starts with a million dollars. Show that the probability that he'll ever be ahead by $2 is less than 1/4.

Explanation

This problem involves determining whether the probability of the gambler being ahead by $2 is less than 1/4. In order to solve this problem, we will need to do the following: 1. Simulate rounds in which the gambler either wins $1 or loses $1 randomly until he reaches $0 or $1,000,002. 2. Simulate the above process many times, and record the results of every game in a vector (1 for win, 0 for loss) 3. Calculate the mean of the results vector in order to determine the actual probability

```
iterations = 1000   #number of rounds
amount = 300    #original balance
resultCounts = vector()   #vector of results
maxValue = amount + 2   #max value of amount before simulation ends
```

In order to accomplish the first step, we will need to initialize variables for his original balance and the number of iterations as shown above. In addition, we will need to create a vector to store the results of each simulation as well as a variable for the max value of amount, which . Although we could hard code these values, for the sake of proper convention, we will be creating variables to maximize customizability. The **amount** variable will be intialized to 300 because initializing it to 1,000,000 would increase the program runtime significantly. In addition, the **iterations** variable will be initialized to 1,000 because we want the number of iterations to be high but not too high such that the program takes too long to run. We will then initialize the **resultCounts** vector, which contains the results of every game, using the *vector()* function. Finally, the **maxValue** variable will be assigned to *amount + 2*, which represents the boundary at which the gambler will be ahead by $2.

```
simulation = function(amt){

  amt = amt + sample(c(1, -1), 1, prob = c(1/3, 2/3))   #generates 1/-1 and adds to amt

  if(amt == maxValue){  #checks to see if the gambler is ahead
    return(1)
  } else if(amt == 0){  #checks to see if the gambler lost all money
    return(0)
  }

  return(simulation(amt))   #recursive call
}
```

After initializing our variables, we can create our *simulation()* function, which takes **amt**, or the amount, as a parameter. The simulation function first uses the *sample()* function to generate either a 1 or a -1 depending on whether the gambler loses or wins a round. This is done randomly, and the *prob* parameter allows us to

modify the probability of winning and losing in order to stay in line with the problem. The resulting value is then added to the existing value of **amt** in order to reflect the change in the gambler's balance. For every call of the *simulation()* function, a check is performed to determine if the gambler's amount is ahead by 2 (using maxValue) or if he has lost all of his money. If either of these conditions holds true, the function returns a 1 or a 0 accordingly (these values will be used to determine the final probability). If the value of **amt** does not meet either of these conditions, it is used in a recursive call to the function.

```
set.seed(6)
for(i in 1:iterations){
  resultCounts[i] = simulation(amount)
}

mean(resultCounts)
```

```
## [1] 0.241
```

After creating the function, a loop is used to run n iterations of the *simulation()* function and assign the resulting output to the i th index of the **resultCounts** vector. By looping this function n times, we are essentially simulating n games. In order to calculate the actual probability, we will need to run many games (in this case 1000), and see how many of those games the gambler actually wins (ends up with $amt + 2$ dollars). Finally, the *mean()* function is used to calculate the probability of the gambler being ahead by $2. Because every "win" is represented as a 1 and every "loss" is represented as a 0, calculating the mean of these results would yield the probability of the gambler being ahead by $2. This holds true for any value of **amount** however, only large values will yield accurate results. This is because the smaller **amount** is, the more random variations dominate the results. After running the program multiple times, we can see that the calculated probability is, in fact, less than 1/4. Although the scope of the problem ends here, we will now write a function to track the results after every round and plot them.

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.3.3
```

```
amountTrack = vector()
```

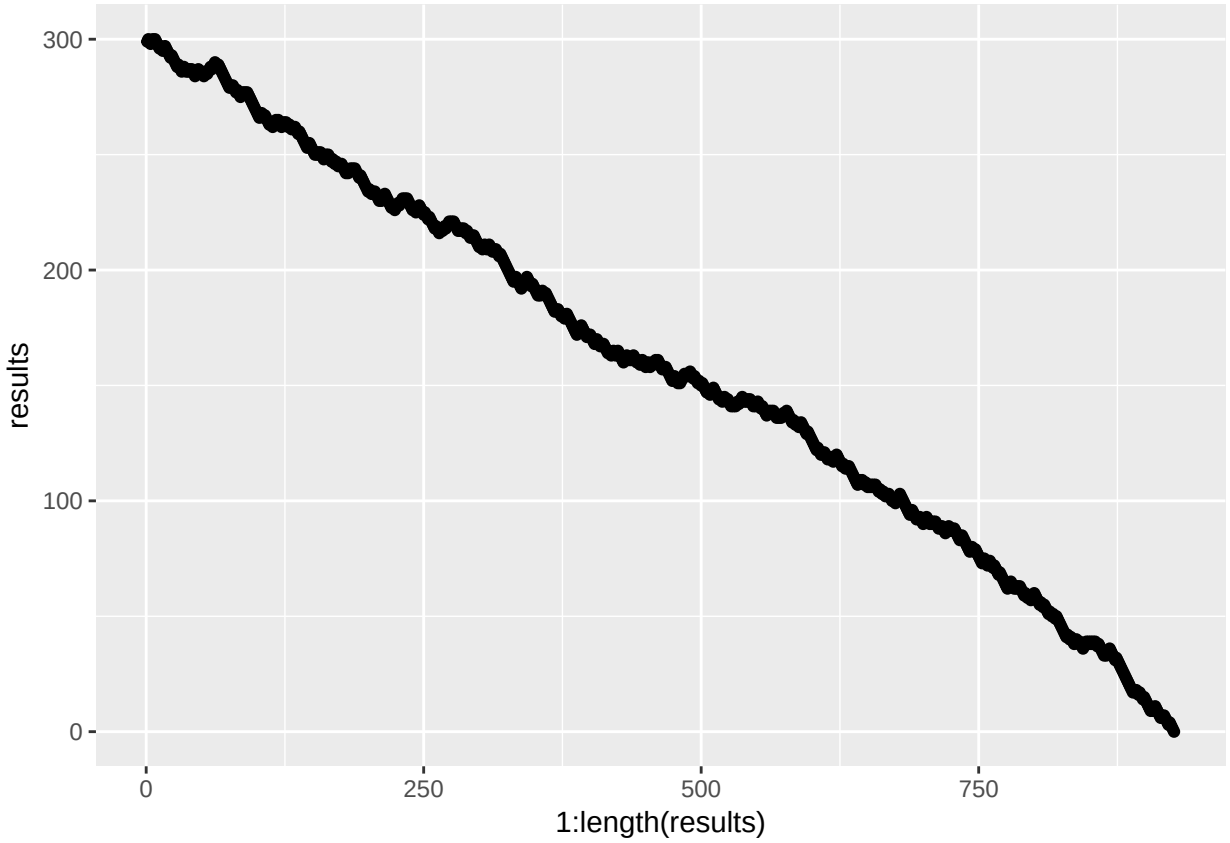
Before writing the function, we first import the *ggplot2* library so that we can plot our results later. We then initialize the **amountTrack** vector using the *vector()* function.

```
trackResults = function(amt, amtTrack){

  amt = amt + sample(c(1, -1), 1, prob = c(1/3, 2/3))   #generates 1/-1 and adds to amt
  amtTrack = c(amtTrack, amt)   #adds new value of amt to amtTrack
  if(amt == maxValue | amt == 0){   #boundary condition
    return(amtTrack)
    }
  return(trackResults(amt, amtTrack))
}
```

We then create the trackResults() function, which takes **amt**, or the amount, as well as the **amtTrack** vector, which tracks the amount after each call of the function. Similar to the *simulation()* function, the *trackResults()* function uses the *sample()* function to randomly generate either a 1 or -1 with varying probability and adds that value to **amt**. We then set **amtTrack** to *c(amtTrack, amt)*, which concatenates the previous values of **amtTrack** and the current value of **amt**. Afterwards, we utilize an *if* statement to determine whether **amt** has reached either boundary (maxValue or 0), in which case the function will return a vector with the values of amt. If **amt** has not reached either boundary value, the function will recursively call itself.

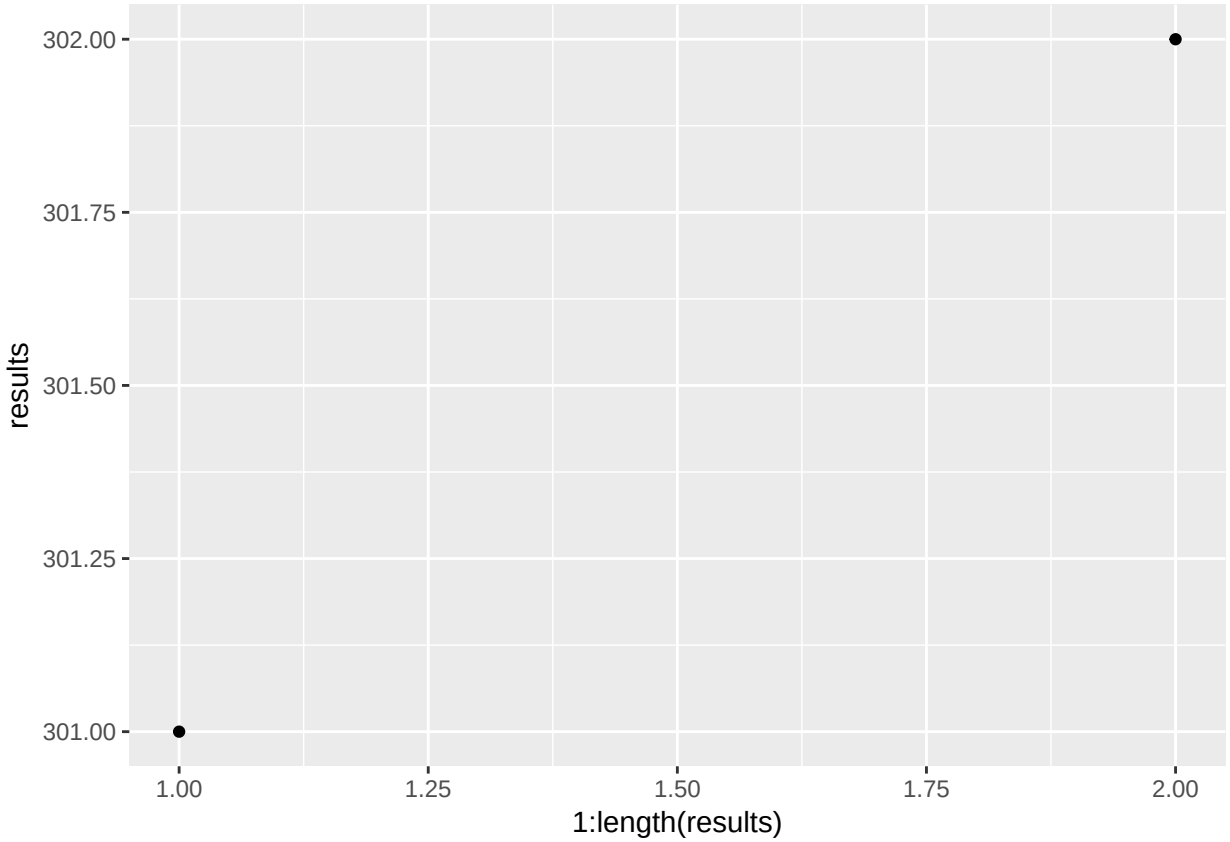
```
set.seed(2)
results = trackResults(amount, amountTrack)
qplot(x = 1:length(results), y = results , geom = "point")
```



After creating the function, we call it and assign the output to a **results** vector. We then use the *qplot()* function to plot the number of rounds as X, and the value of **amt** as Y.

In order to easily view the two different plots this program can output, we must use the *set.seed()* function to set the seed for our program. If we input the number *2*, the resulting plot will show the gambler as he loses all of his money.

```
set.seed(78)
results = trackResults(amount, amountTrack)
qplot(x = 1:length(results), y = results , geom = "point")
```



If we input the number *78*, the resulting plot will show the gambler be ahead by 2. It is clear that the second plot shows significantly fewer rounds and this is due to the fact that it is much more probable for the gambler to be ahead by two after only a few rounds, rather than losing many rounds and then going all the way back to **maxValue** or *amt + 2*.

Conclusion

Thank you for taking the time to read this explanation for the "Gambler's Ruin" problem from "Statistics 110: Strategic Practice & Homework 3", courtesy of Harvard University. Link: https://projects.iq.harvard.edu/files/stat110/files/strategic_practice_and_homework_3.pdf. This problem was taken from the public resources to the course and **is not my orginal work**. The explanation, however, is 100% original. Please feel free to visit my GitHub page at http://github.com/goutham1220 where I will be posting more explanations as well as other statistics and data science-related resources. In addition, please feel free to visit my YouTube channels "GSDataScience" (http://bit.ly/gsdatascience), where I will be posting more data science and statistics-related videos, and "Gooth" (http://youtube.com/gooth), where I post more cinematic-style, slice-of-life videos.