Gavin Bolin

Probability LAB

1) For this first lab, I wrote a small program to run a while loop to fail if a random number between 0 and 1 is less than 0.5. Then I would call this as many times as I wanted to simulate the test and, in the end, it would return the average money made per iteration as well as the most money made in a single iteration. Below are the results to 4 different runs of the 3, 100, 10000, and 1000000 iteration tests.

Average$/game Most$/game	Iterations			
Attempt	3	100	10000	1000000
1	$0.2333 $4	$3.76 $128	$7.379 $4096	$10.4625 $524288
2	$11.3333 $32	$5.16 $256	$6.5219 $4096	$11.0666 $1048576
3	$43.6666 $128	$6.12 $256	$7.7529 $4096	$10.3878 $1048576
4	$22.6666 $64	$3.85 $32	$7.2775 $4096	$10.0846 $524288

The more times that you play the game the higher the amount of money you can potentially get back per iteration from the game. It's interesting that each time the game is run for 1000000 iterations it comes out as $10/game on average. Therefore, the more games that are played the higher chance you have to get a run that will get you more money. I would pay as much as I can if that correlates to more runs because the more iterations that are run the more money per round you are bound to win. However, if I only got to play once I would pay $2 to try it and even if I lost, I would still get half of money back but have the potential like in run 3 with 3 iterations, $42 were won out of 3 rounds and even in the million iteration there was one where you could win a million dollars.

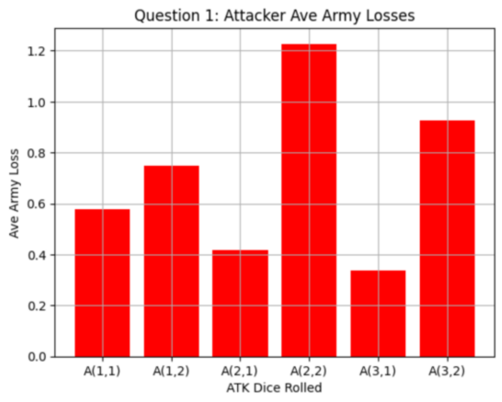
2) For this game it was hard to tell if the door that was opened after the goat was revealed could be the same as the one chosen by the player originally, but I was able to make it that way. The win rate of staying with the original door and the win rate of switching doors was very close. However more times than not the win rate after 1000 iterations was better after switching than without switching. The odds of either should still be around a third even though you are basically only choosing from 2 doors.

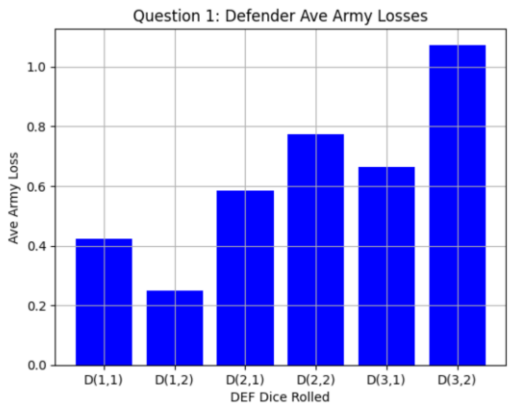
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
Switching: True	%33.5	%33.4	%32.3	%34
Switching: False	%31.8	%31.1	%35	%33.1

3)

Part 1:

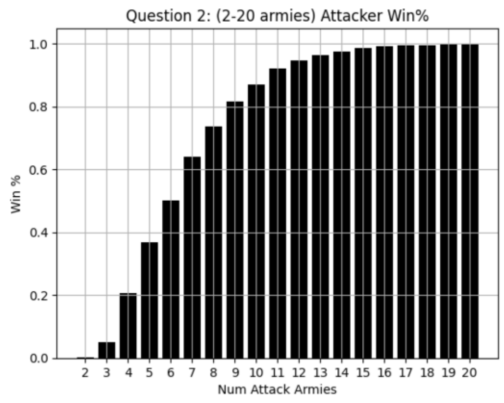
I used a test of 10000 trials to be able to have a broad enough sample for reliable data but without overdoing it as the method for calculating this was a triple nested for loop. However, the first loop was n while in each of those iterations were 6 more iterations for each combination of the dice rolled by each. By reviewing the data when rolling as the attacker the highest win rate, or conversely the least loss with rolling combination was with 3 dice while the defender has 1. But very close behind it was only 2 dice if the defender only has one die. The defender is usually at an advantage as they can lose both dice when starting with 2 but the attacker can only lose 2 when starting at 3. The highest win rate is found when the defender is at an advantage of number of dice. But as we increase in dice number it gets harder to win. The next best win rate for the defender is found when both players have 1 die.





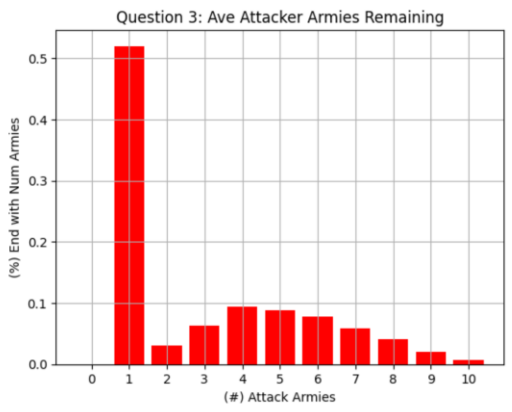
Part2:

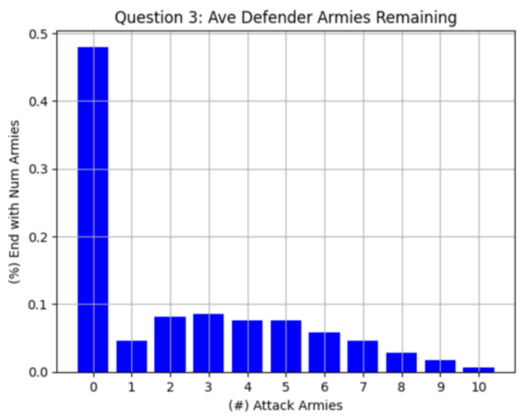
We can see the clear favor toward more armies correlating to higher win percentage for attackers. This is given that the defender has only 5 armies at the location. The amount of iterations as set at 10000 again for this problem and took the longest to generate at each iteration runs 19 times (2-20 armies) even though it is only calculating for the attacker. We reach nearly 100% victory as we approach 17-19 armies even with 10000 iterations. To be able to ensure a 50%-win rate we need at least 6 armies. To ensure an 80%-win rate we need at least 9 armies, and the graph shows that we can confidently say that as it is just past 80%.



Part3:

This graph shows how often the battle ends with the certain number of armies by each side. We can see that based on the rate of 1 army left of the attackers and 0 armies let by the defenders we can tell how often they lost. About 52% of battles end with the defender winning and 48% end with the attacker winning. However the data suggests the average number of armies left after the battle tends to lean on the side of the attacker with a rough mean around 4 and a rough mean of the defender around 3.





INDEX

Problem 1:

```
#ST PETERSBURG PARADOX:
import numpy as np
import random as r

def iteration():
  count = coin_toss()
  total = pow(2,count)
  return total

def coin_toss():
  heads = True
  count = 0
  while heads:
    rand = r.random()
    if rand < .5:
      heads = False
    else:
      count +=1
  return count

def peter(n):
  most = 0
  total = 0
  for i in range(n):
    it_total = iteration()
    if(it_total > most): most = it_total
    total += it_total
  average = total / n
  print('Average$/game:', average,'\nMost$/game:', most, '\n')

peter(3)
peter(100)
peter(10000)
peter(1000000)
```

Problem 2:

```
#MONTY HALL PROBLEM

def iteration(switch):
  avail = ['door1','door2','door3']
  car = generalize(r.random())
  avail.remove(car)
  guess = generalize(r.random())
  for i in avail:
    if i == guess:
      avail.remove(guess)
  goat = r.random()
  if avail.count == 2:
    if goat < .5:
      if (switch): guess = avail[1]
    else:
      if (switch): guess = avail[0]
  #print(guess, ' ', car, '\navail:', avail, '\ngoat:', goat)
  if guess == car: return True
  else: return False

def generalize(x):
  if x < .3333:
    return 'door1'
  elif x > .6666:
    return 'door3'
  else:
    return 'door2'

def hall(n, switch):
  total = 0
  for i in range(n):
    win = iteration(switch)
    if win == True: total +=1
  ave_acc = total / n
  print('Switch after goat revealed:', switch, '\nAverage winrate:',
ave_acc)

hall(1000, True)
hall(1000, False)
```

Problem 3:

```
#RISK BATTLE

def battle(attack,defend):
  while defend > 0 and attack > 1:
    a_roll_n = 0
    d_roll_n = 0
    if attack > 3: a_roll_n = 3
    elif attack == 3: a_roll_n = 2
    else: a_roll_n = 1
    if defend > 1: d_roll_n = 2
    else: d_roll_n = 1
    atk_res, def_res = fight(a_roll_n,d_roll_n)
    attack -= atk_res
    defend -= def_res
  return attack, defend

def fight(a_roll_n, d_roll_n):
  aq,dq = [], []
  for i in range(a_roll_n):
    aq.append(dice(r.random()))
  for i in range(d_roll_n):
    dq.append(dice(r.random()))
  aq.sort(reverse=True)
  dq.sort(reverse=True)
  a_loss,d_loss = 0,0
  least_n = min(len(aq),len(dq))
  for i in range(least_n):
    a=aq.pop(0)
    d=dq.pop(0)
    if(a > d):
      d_loss+=1
    elif(a <= d):
      a_loss+=1
  return a_loss, d_loss

def dice(x):
  if x < .5:
    if x < .1666:return '1'
    elif x > .3333: return '3'
    else: return '2'
  else:
    if x < .6666: return '4'
    elif x > .8333: return '6'
    else: return '5'

def graph(title, x, y, names, res, color):
  plt.title(title)
  plt.bar(names, res, color=color)
  plt.xlabel(x)
  plt.ylabel(y)
  plt.grid()
  plt.show()

def q1(n):
  res = [0]*12
  for iter in range(n):
    ain,din=0,6
    for i in range(3):
      for j in range(2):
        atk_res, def_res = fight(i+1,j+1)
        res[ain] = res[ain]+atk_res
        res[din] = res[din]+def_res
        ain,din = ain+1,din+1
  for i in range(len(res)):
      res[i] = res[i] / n
  graph('Question 1: Attacker Ave Losses',
        'ATK Dice Rolled',
        'Ave Losses',
        ['A(1,1)', 'A(1,2)', 'A(2,1)', 'A(2,2)', 'A(3,1)', 'A(3,2)'],
        res[:6], color='red')
  graph('Question 1: Defender Ave Losses',
        'DEF Dice Rolled',
        'Ave Losses',
        ['D(1,1)', 'D(1,2)', 'D(2,1)', 'D(2,2)', 'D(3,1)', 'D(3,2)'],
        res[6:], color='blue')

def q2(n):
  total = 0
  res = [0]*19
  for iter in range(n):
    for i in range(2,21):
      atk_res, def_res = battle(i,5)
      if atk_res > 1: win = 1
      else: win = 0
      res[i-2] = res[i-2] + win
  for i in range(len(res)): res[i] = res[i] / n
  print(res)
  graph('Question 2: (2-20 armies) Attacker Win%',
        'Num Attack Armies',
        'Win %',
        ['2','3','4','5','6','7','8','9','10','11','12','13','14','15','16
','17','18','19','20'],
        res, color='black')

def q3(n):
  atk_rem = [0]*11
  def_rem = [0]*11
  for i in range(n):
    atk_res, def_res = battle(10,10)
    atk_rem[atk_res] = atk_rem[atk_res] + 1
    def_rem[def_res] = def_rem[def_res] + 1
  print(atk_rem)
  for i in range(len(atk_rem)):
    atk_rem[i] = atk_rem[i] / n
    def_rem[i] = def_rem[i] / n
  print(atk_rem, def_rem)
  graph('Question 3: Ave Attacker Armies Remaining',
        '(#) Attack Armies',
        '(%) End with Num Armies',
        ['0','1','2','3','4','5','6','7','8','9','10'],
        atk_rem, color='red')
  graph('Question 3: Ave Defender Armies Remaining',
        '(#) Attack Armies',
        '(%) End with Num Armies',
        ['0','1','2','3','4','5','6','7','8','9','10'],
        def_rem, color='blue')

q1(10000)
q2(10000)
q3(20000)
```