<Probability Estimation>

1. St. Petersburg Paradox

```python
import random

# St. Petersburg Paradox

# k tosses until see the first tail -> 2^k dollars
# N : number of times to play the game
def st_ptb_paradox(N):

    total_money_earned = 0
    most_money_earned = 0
    for i in range(N):
        money_earned = 0
        tosses = 0
        while True:
            # 0 is head , 1 is tail
            head_or_tail = random.randint(0, 1)
            tosses = tosses + 1

            # tail first came up
            if head_or_tail == 1:
                money_earned = 2 ** tosses
                if money_earned > most_money_earned:
                    most_money_earned = money_earned
                break

        total_money_earned = total_money_earned + money_earned

    print("total : $", total_money_earned)
    print("Most money earned : $", most_money_earned)
    print("Average money earned : $", total_money_earned / N, "\n")

st_ptb_paradox(100)
st_ptb_paradox(10000)
st_ptb_paradox(1000000)
```

total : $ 534

Most money earned : $ 64

Average money earned : $ 5.34

total : $ 227,342

Most money earned : $ 65,536

Average money earned : $ 22.7342

total : $ 53,613,186

Most money earned : $ 33,554,432

Average money earned : $ 53.613186

The expected value for this game approaches infinity, but it seems not rational to pay that amount of money to play this game because the average amount of money we can get is pretty low. I don't think I will be willing to pay more than $10 to play this game even though what I can ideally expect to get is almost infinite in theory.

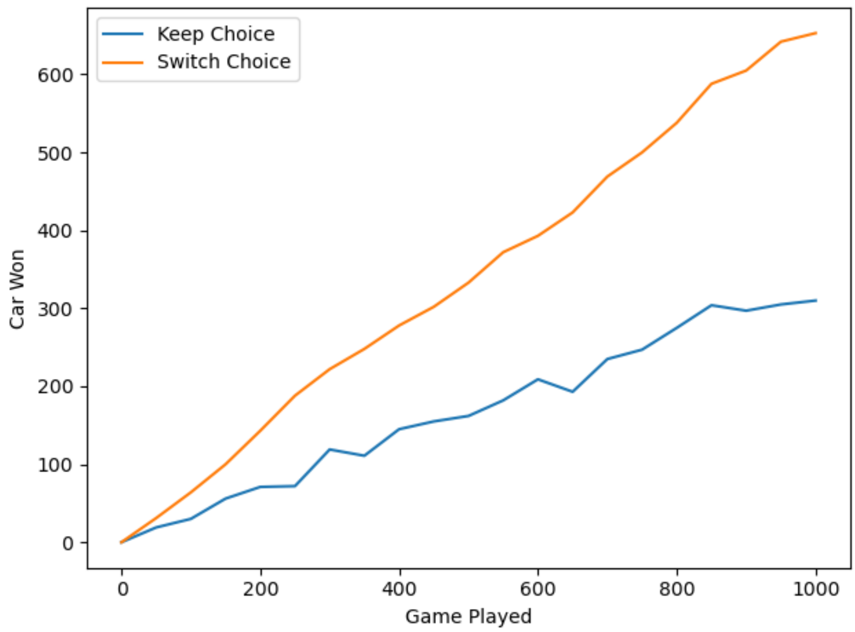
2. Monty Hall Problem

```
import random
import matplotlib.pyplot as plt

# Play N times
def simulate_monty_hall(N, keep_choice):

    keep_strategy = True
    if keep_choice == False:
        keep_strategy = False

    total_car_won = 0

    for i in range(N):

        three_doors = ['goat', 'goat', 'goat']

        # Randomly assign car's location
        car_loc = random.randint(0, 2)
        three_doors[car_loc] = 'car'

        guess = random.randint(0, 2)

        rest_loc = []
        for j in range(len(three_doors)):
            if j != guess:
                rest_loc.append(j)

        door_to_open = -1
        # Randomly pick the door with goat
        while True:
            ran_num = random.randint(0, 1)
            if three_doors[rest_loc[ran_num]] == 'goat':
                door_to_open = rest_loc[ran_num]
                del rest_loc[ran_num]
                break

        three_doors[door_to_open] = 'goat-revealed'

        # Keep the first selection
        if keep_strategy == True:
            if three_doors[guess] == 'car':
                total_car_won = total_car_won + 1

        # Switch the choice
        else:
            door_to_change = rest_loc[0]
            if three_doors[door_to_change] == 'car':
                total_car_won = total_car_won + 1


    return total_car_won

def main():
```

```
    x = []
    y_keep = []
    y_switch = []

    for i in range(0, 1001, 50):
        x.append(i)
        y_keep.append(simulate_monty_hall(i, True))
        y_switch.append(simulate_monty_hall(i, False))

    plt.plot(x, y_keep, label = "Keep Choice")
    plt.plot(x, y_switch, label = "Switch Choice")
    plt.xlabel("Game Played")
    plt.ylabel("Car Won")
    plt.legend()
    plt.show()

    print("Percentage of keeping choice = ", (simulate_monty_hall(1000, True) / 1000) * 
100)
    print("Percentage of switching choice = ", (simulate_monty_hall(1000, False) / 
1000) * 100)

if __name__ == "__main__":
    main()
```

Percentage of keeping choice = 33.2

Percentage of switching choice = 66.2



When playing 1000 times, the percentage if the player kept his first choice was 33.2%, and 66.2% when he switched his choice. Also, as we can see in the graph above, it's clear to see the difference when the game was played below 50 times. When a player makes his decision at first, the probability would be almost 33% because the prize would be in one of the three doors. This can be said that when he makes his first choice, the probability that the door chosen is incorrect is almost 66%. Since the host eliminates one of the incorrect doors, it's better to switch the door because it has 66% of winning the car.

3. RISK Battle

```python
import random
import matplotlib.pyplot as plt
import numpy as np

def get_prob(na, nd, N):

    total_attack_army_lost = 0
    total_defend_army_lost = 0

    attacker_lost_no_army = 0
    defender_lost_no_army = 0
    attacker_lost_one_army = 0
    attacker_lost_two_army = 0
    defender_lost_one_army = 0
    defender_lost_two_army = 0

    # N times to play
    for i in range(N):

        attack_army_lost = 0
        defend_army_lost = 0

        attack_dice_result = []
        defend_dice_result = []

        for j in range(na):
            attack_dice_result.append(random.randint(1, 6))
        for j in range(nd):
            defend_dice_result.append(random.randint(1, 6))

        # sort the result
        attack_dice_result.sort(reverse=True)
        defend_dice_result.sort(reverse=True)

        # Match up
        match_count = min(na, nd)
        for j in range(match_count):
            if attack_dice_result[j] > defend_dice_result[j]:
                defend_army_lost = defend_army_lost + 1
            else:
                attack_army_lost = attack_army_lost + 1

        if defend_army_lost == 1:
            defender_lost_one_army = defender_lost_one_army + 1
        if defend_army_lost == 2:
            defender_lost_two_army = defender_lost_two_army + 1
        if attack_army_lost == 1:
            attacker_lost_one_army = attacker_lost_one_army + 1
        if attack_army_lost == 2:
            attacker_lost_two_army = attacker_lost_two_army + 1
        if defend_army_lost == 0:
            defender_lost_no_army = defender_lost_no_army + 1
        if attack_army_lost == 0:
            attacker_lost_no_army = attacker_lost_no_army + 1

        total_attack_army_lost = total_attack_army_lost + attack_army_lost
        total_defend_army_lost = total_defend_army_lost + defend_army_lost

    print("For attacker, losing nothing : ", attacker_lost_no_army / N)
    print("For attacker, losing one : ", attacker_lost_one_army / N)
    print("For attacker, losing two : ", attacker_lost_two_army / N, "\n")

    print("For defender, losing nothing : ", defender_lost_no_army / N)
```

```
    print("For defender, losing one : ", defender_lost_one_army / N)
    print("For defender, losing two : ", defender_lost_two_army / N)

    print((defender_lost_no_army + defender_lost_one_army + defender_lost_two_army) /
N)

def get_prob_2(a_army, d_army, N):

    attacker_win = 0
    remaining_a_armies = []
    remaining_d_armies = []

    for i in range(N):

        attacker_army = a_army
        defender_army = d_army

        while True:

            a_dice = 3
            d_dice = 2

            attack_dice_result = []
            defend_dice_result = []

            if defender_army == 1:
                d_dice = 1
            if attacker_army == 3:
                a_dice = 2
            elif attacker_army == 2:
                a_dice = 1

            for j in range(a_dice):
                attack_dice_result.append(random.randint(1, 6))
            for j in range(d_dice):
                defend_dice_result.append(random.randint(1, 6))

            # sort the result
            attack_dice_result.sort(reverse=True)
            defend_dice_result.sort(reverse=True)

            # Match up
            match_count = min(a_dice, d_dice)
            for j in range(match_count):
                if attack_dice_result[j] > defend_dice_result[j]:
                    defender_army = defender_army - 1
                else:
                    attacker_army = attacker_army - 1

            # attacker wins
            if defender_army == 0:
                attacker_win = attacker_win + 1
                remaining_a_armies.append(attacker_army)
                break
            if attacker_army == 1:
                remaining_d_armies.append(defender_army)
                break


    #return attacker_win / N
    return remaining_a_armies, remaining_d_armies


def main():
    remaining_a_armies, remaining_d_armies = get_prob_2(10, 10, 10000)
    np_a_armies = np.array(remaining_a_armies)
    np_d_armies = np.array(remaining_d_armies)
```

```
    # when attacker won
    x_a = [i for i in range(2, 11)]
    y_a = []
    for i in range(2, 11):
        y_a.append((np_a_armies == i).sum())

    x_d =[i for i in range(1, 11)]
    y_d = []
    for i in range(1, 11):
        y_d.append((np_d_armies == i).sum())

    plt.plot(x_d, (np.array(y_d) / len(remaining_d_armies)))
    plt.xlabel("Number of Remaining Defender's Army")
    plt.ylabel("Probability")

    #plt.plot(x_a, (np.array(y_a) / len(remaining_a_armies)))
    #plt.xlabel("Number of Remaining Attacker's Army")
    #plt.ylabel("Probability")
    plt.show()



def solve_2():
    x = [i for i in range(2, 21)]
    y = []

    for i in range(2, 21):
        y.append(get_prob_2(i, 10000))
        # print("Number of attacker's army : ", i, " and winning probability : ",
get_prob_2(i, 5, 10000))

    plt.plot(x, y)
    plt.xlabel("Number of attacker's army")
    plt.ylabel("Attacker-win-probability")
    plt.show()

if __name__ == "__main__":
    main()
```

1) Probabilities / advantageous?

<When attacker rolls 3 dice, and the defender rolls 2 dice>

- Attacker loses nothing, defender loses two : 0.3772
- Attacker loses one, defender loses one : 0.3331
- Attacker loses two, defender loses nothing : 0.2897

<When attacker rolls 3 dice, and the defender rolls 1 dice>

- Attacker loses nothing, defender loses one : 0.6673
- Attacker loses one, defender loses nothing : 0.3327

<When attacker rolls 2 dice, and the defender rolls 2 dice>

- Attacker loses nothing, defender loses two : 0.2278

- Attacker loses one, defender loses one : 0.3248
- Attacker loses two, defender loses two : 0.2278

<When attacker rolls 2 dice, and the defender rolls 1 dice>

- Attacker loses nothing, defender loses one : 0.5788
- Attacker loses one, defender loses nothing : 0.4212

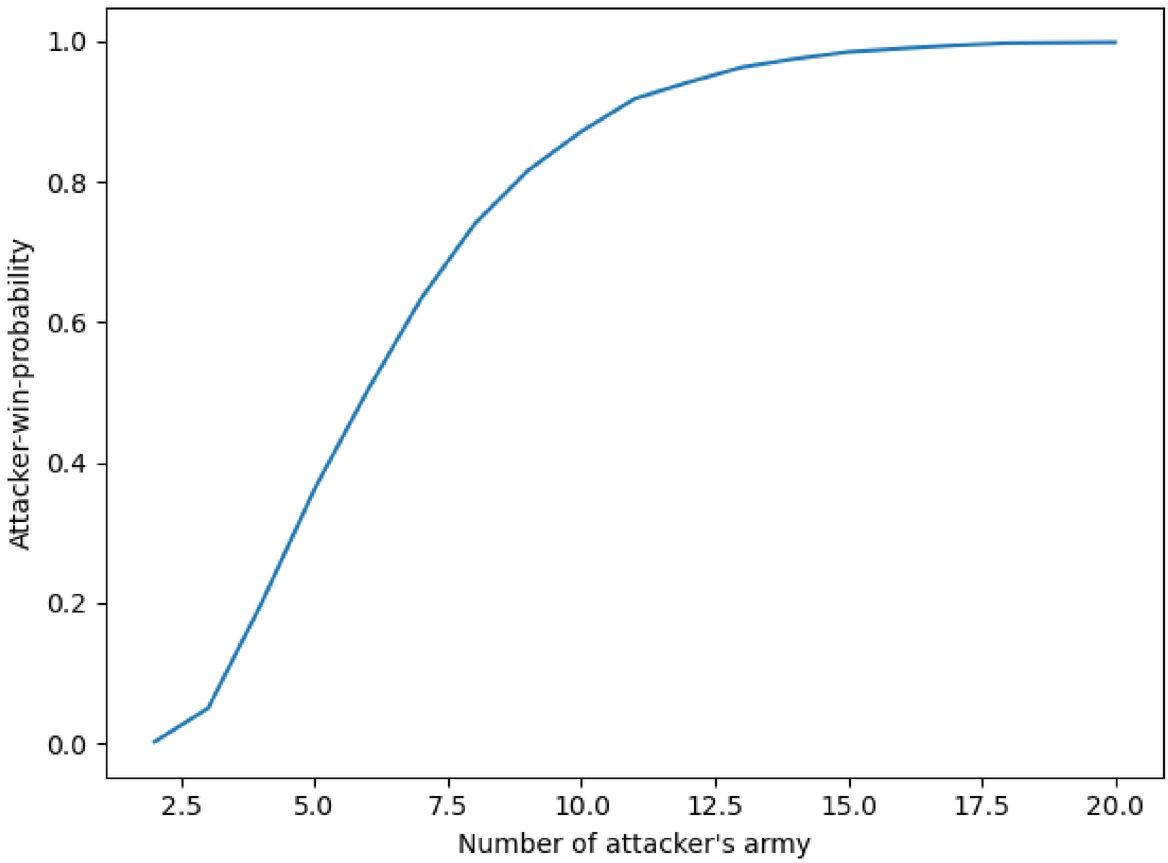
<When attacker rolls 1 dice, and the defender rolls 1 dice>

- Attacker loses nothing, defender loses one : 0.4103
- Attacker loses one, defender loses nothing : 0.5879

<When attacker rolls 1 dice, and the defender rolls 2 dice>

- Attacker loses nothing, defender loses one : 0.251
- Attacker loses one, defender loses nothing : 0.749

I used 10,000 samples for each of the cases since the results seemed consistent after that number. According to the result, it is not advantageous for a player to roll less than the most dice they are allowed because it would lower the probability of winning. By rolling more dice, we have better probability that we would see higher numbers from the dice.



2) I used the same 10,000 samples so that the graph would be smooth and that the result is more consistent.

Number of attacker's army : 2 and winning probability : 0.0017

Number of attacker's army : 3 and winning probability : 0.0481

Number of attacker's army : 4 and winning probability : 0.208

Number of attacker's army : 5 and winning probability : 0.3519

Number of attacker's army : 6 and winning probability : 0.503

Number of attacker's army : 7 and winning probability : 0.6369

Number of attacker's army : 8 and winning probability : 0.7323

Number of attacker's army : 9 and winning probability : 0.8164

Number of attacker's army : 10 and winning probability : 0.8734

Number of attacker's army : 11 and winning probability : 0.9174

Number of attacker's army : 12 and winning probability : 0.9398

Number of attacker's army : 13 and winning probability : 0.9629

Number of attacker's army : 14 and winning probability : 0.9765

Number of attacker's army : 15 and winning probability : 0.9819

Number of attacker's army : 16 and winning probability : 0.9879

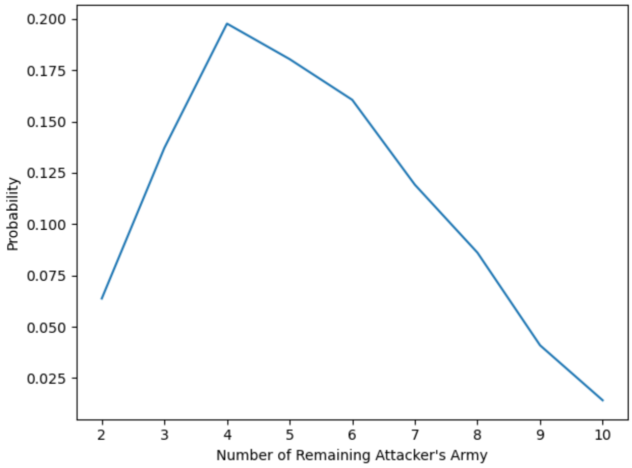
Number of attacker's army : 17 and winning probability : 0.9943

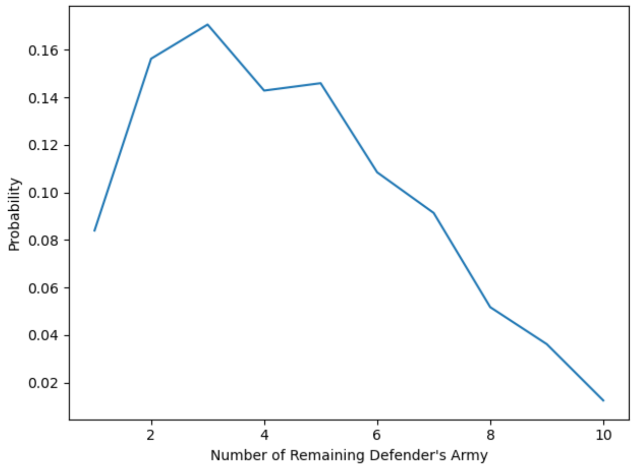
Number of attacker's army : 18 and winning probability : 0.9961

Number of attacker's army : 19 and winning probability : 0.9968

Number of attacker's army : 20 and winning probability : 0.9985

It seems the attacker needs at least 6 armies in order to guarantee a 50% chance of winning the territory from this sampling case. Also, the attacker needs at least 9 armies to guarantee 80% chance of winning.





3) Each of this table shows the probability for each possible number of remaining armies of each player at the end of the battle. I used the sample number of 10,000 to simulate the game and got this result. This shows that it is most likely for the attacker to have 3 – 6 armies remaining at the end of the game if the attacker won the battle. Similarly, the defender would likely to have 2 – 5 armies remaining if the defender won the game.