Texas Hold-Em Poker Calculator

By Daniel

What are we going to talk about

- What are we trying to answer within poker
- What I did to approach the problem
- Go through important parts in the codebase and its use
- Show examples where it can be used to help us answer our questions.

What is our problem?

- Poker is a card game where players wager over **which hand is best**.
- As a player of poker, you want to be able to win more than you lose.
- So what can we do in order to **enrich someone's experience of the game?**
- How likely are we to win a poker game?
- How does this change in different situations? (different players/hands)
- What are some sure signs that you are in a good or bad place throughout the game?
- What are the different mathematical approaches and how can we do it computationally. How intensive is it to calculate that?

Rules Used

We are playing Texas Holdem poker:

- Each player is dealt **2 cards** at the beginning
- A total of **5 cards** used by everyone are dealt throughout the game over **3 rounds**
- The first round (Flop) reveals **3 cards**
- Then the second reveals another leaving **4 cards**
- And then there's the final round before the reveal which shows a total of **5 cards**
- This leaves each player with **7 cards** to make a **5 card combination** for the best hand possible

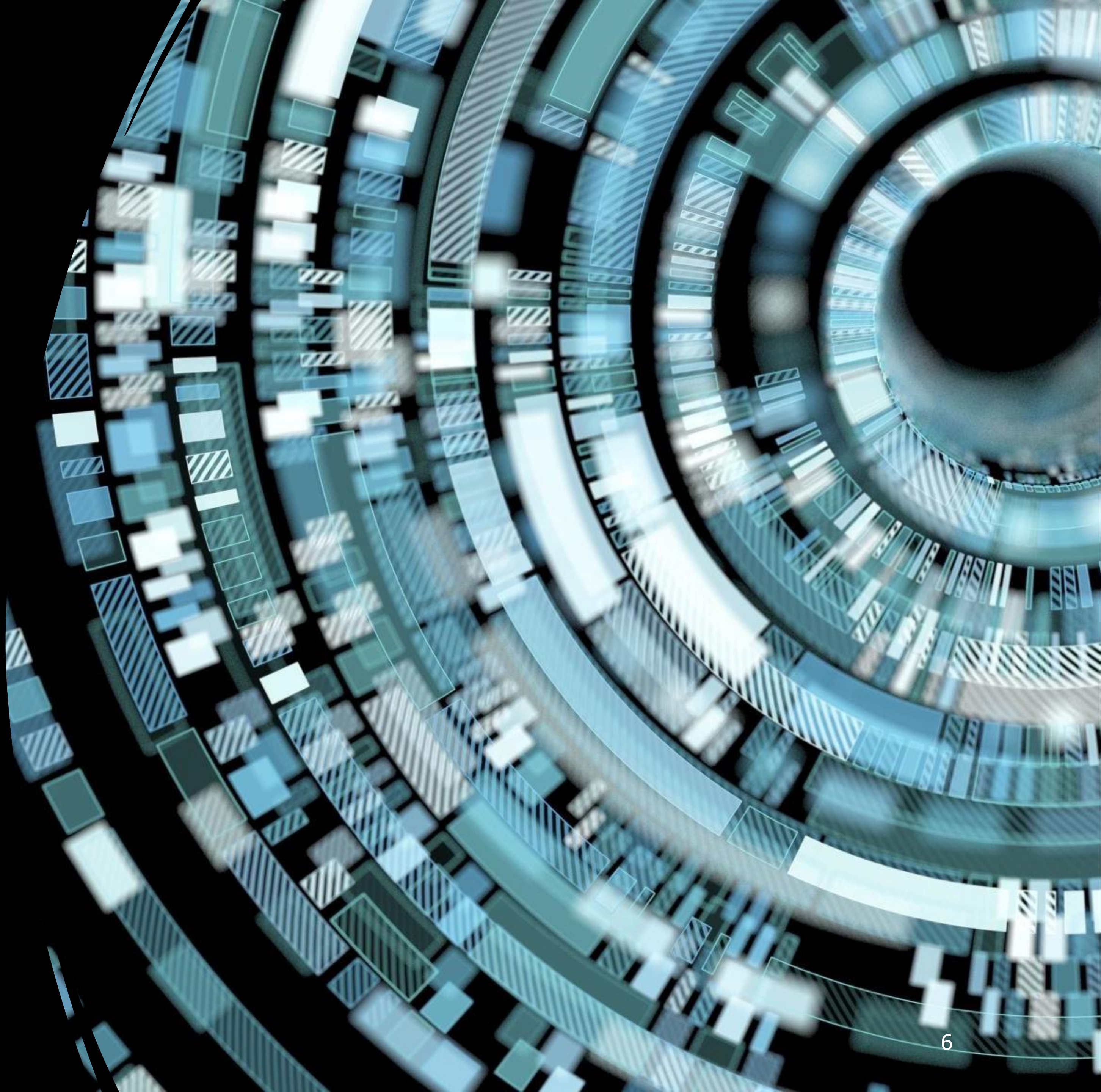
- Poker typically includes the ability of betting and folding (leaving the game), but we left this out to reduce complexity.

Solution

- I programmed:
  - A virtual engine that runs Texas hold-Em poker games
    - A **data table** containing all poker hands and their rank (with tiebreakers) self **generated by a combinations algorithm.**
    - A poker comparer **that uses the data table** to find the strongest poker hand and compares poker hands with other opponents (to find the winner)
    - A **GUI** (Graphical Interface) to help understand the information presented throughout the game
  - An **algorithm that looks through future poker hand combinations** to forewarn and predict future outcomes while playing
  - A **Monte Carlo Simulator** of Poker that allow for games of poker to be run instantaneously that can allow for experimenting with different situations.

What Techniques and Libraries?

- Binomial coefficients – Itertools, maths
- Object oriented programming
- Data manipulation (CSV) – Pandas Numpy (Pickle)
- GUI – Pygame
- Monte Carlo Simulations – random
- Visuals – Power BI

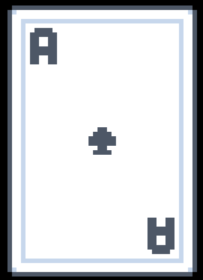


Creating the base program

Poker Entities using OOP

```python
class Card():
    suits = ["hearts", "diamonds", "clubs", "spades"]
    values = ["2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"]

    def __init__(self,properties,parent):
        self.suit = properties[-1]
        self.value = properties[:-1]
        self.cardval = properties
        self.parentchange(parent)
        #print(f" {self.value} {self.suit} Card:  {self.cardval} ")

    def parentchange(self,newparent):
        self.parent = newparent
        self.place = newparent.cards
```

Card Object



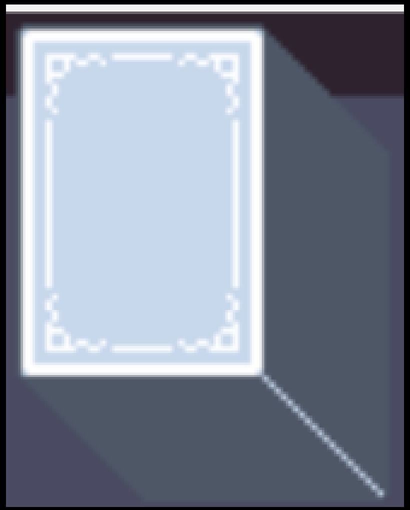
Used to store card information and states (like location)

```python
class Deck():
    deck = ['2♥', '2♦', '2♣', '2♠', '3♥', '3♦', '3♣', '3♠',
        '4♥', '4♦', '4♣', '4♠', '5♥', '5♦', '5♣', '5♠',
        '6♥', '6♦', '6♣', '6♠', '7♥', '7♦', '7♣', '7♠',
        '8♥', '8♦', '8♣', '8♠', '9♥', '9♦', '9♣', '9♠',
        '10♥', '10♦', '10♣', '10♠', 'j♥', 'j♦', 'j♣', 'j♠',
        'q♥', 'q♦', 'q♣', 'q♠', 'k♥', 'k♦', 'k♣', 'k♠',
        'a♥', 'a♦', 'a♣', 'a♠']

    def __init__(self):
        self.name = "deck"
        self.cards = []

    def shuffle(self):
        # Shuffle the cards in the deck
        random.shuffle(self.cards)

    def deckGen(self):
        # Generate the deck with Card objects and shuffle
        for i in Deck.deck:
            self.cards.append(Card(i,self))
        self.shuffle()

    def getCardPlace(self,card):
        # Get the index of the specified card in the deck
        return self.cards.index(card)

    def getCardPlace2(self,card):
        index = [i for i, cardv in enumerate(self.cards) if cardv.cardval == card]
        return index[0]

    def drawCard(self,location):
        # Draw the top card from the deck and move it to the specified location
        self.move(self.cards[-1],location)

    def move(self,card,location):
        # Move the specified card to the specified location
        if self.cards.count(card) > 0:
            card.parentchange(location)
            if location == self:
                self.cards.append(card)
            else:
                location.cards.append(card)
            self.cards.remove(card)
```

Deck Object

Used to store cards and has special functions that help to simulate a game of poker accurately

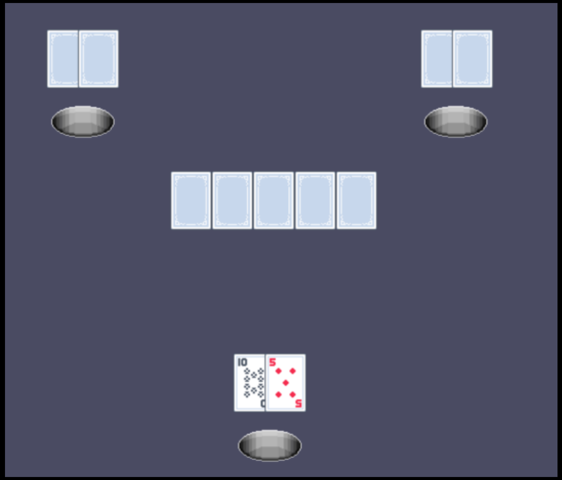
(e.g. Shuffle, draw card)



```python
class Hand():
    Hands = 0
    def __init__(self):
        self.name = "hand"
        self.cards = []
        Hand.Hands += 1

    def move(self,card,location):
        if self.cards.count(card) > 0:
            card.parentchange(location)
            if location == self:
                self.cards.append(card)
            else:
                location.cards.append(card)
            self.cards.remove(card)
        else:
            pass

    def getCardPlace(self,card):
        return self.cards.index(card)
    def getCardPlace2(self,card):
        index = [i for i, cardv in enumerate(self.cards) if cardv.cardval == card]
        return index[0]


class EHand(Hand):
    def init(self):
        super(Hand,self).__init__(self)


class Community():
    def __init__(self):
        self.name = "community"
        self.cards = []

    def getCardPlace(self,card):
        return self.cards.index(card)
    def getCardPlace2(self,card):
        index = [i for i, cardv in enumerate(self.cards) if cardv.cardval == card]
        return index[0]

    def move(self,card,location):
        if self.cards.count(card) > 0:
            card.parentchange(location)
            if location == self:
                self.cards.append(card)
            else:
                location.cards.append(card)
            self.cards.remove(card)
        else:
            pass
```

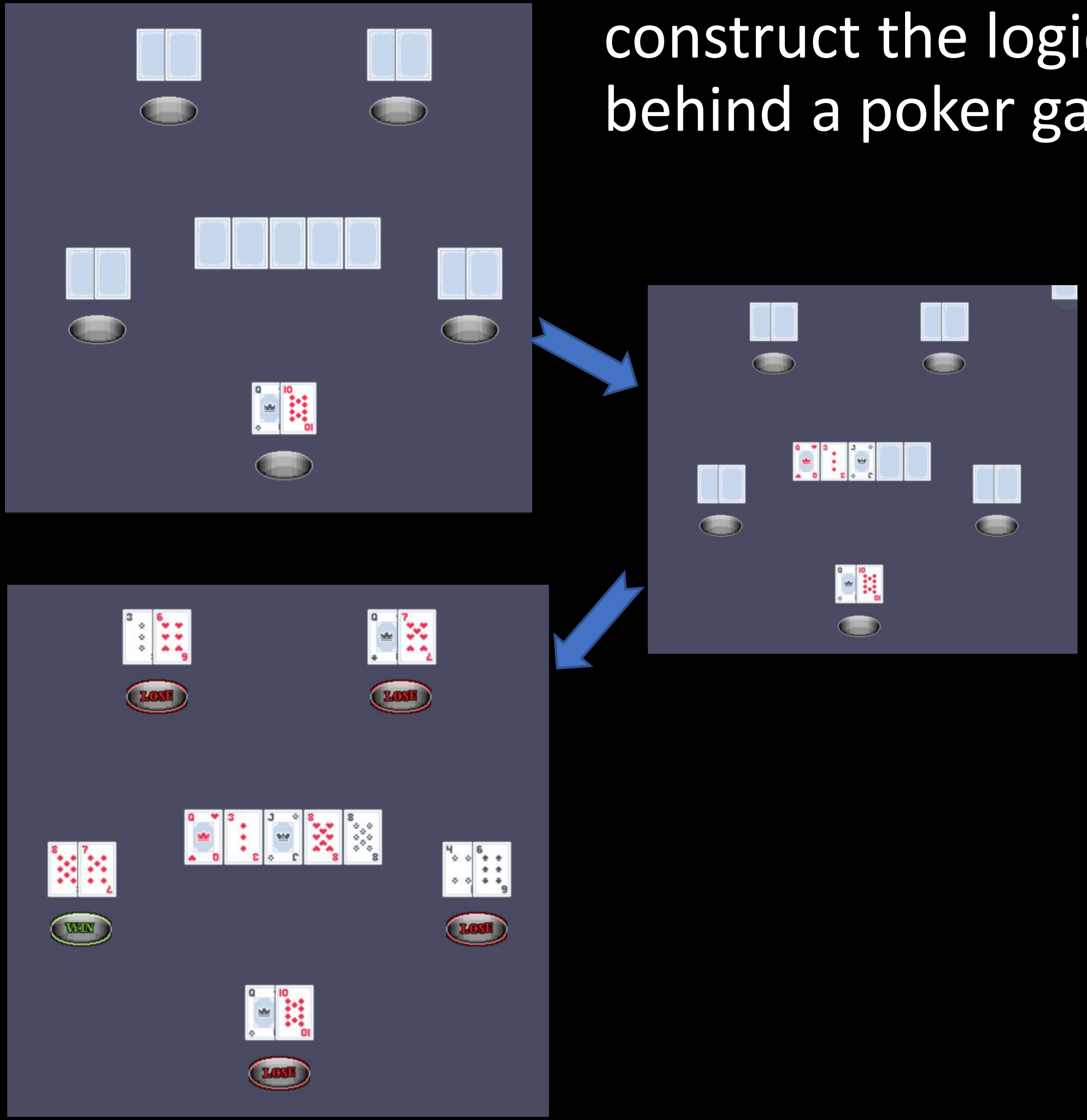
Hand and Community Objects

Both are very similar to the deck in functionality since it also stores cards in the same way.



With this, we can construct the logic behind a poker game

```python

    def draw(self,playerhand,otherhands,deck,community): ···

    def grabPlayerHand(self): ···
    def grabCommunityHand(self): ···

    def grabEvaluation(self): ···

    def finalEvaluation(self): ···


    def revealWin(self,winner=None): ···


    def resetWin(self):        You, 2 seconds ago • Uncommitted changes ···

    def phaseGame(self):
        if self.phase == 1:
            print("\n\n\n\n\n\n\n____________Pre Flop____________")
            self.part = "Pre Flop"
            self.draw(*self.groups)
            self.grabPlayerHand()
            self.grabCommunityHand()
            self.sound.soundPlay("round")
            self.rev = 0


        if self.phase == 2:
            print("\n\n\n\n\n\n\n____________Flop____________")
            self.part = "Flop"
            self.groups[3].flipCards((0,3))
            self.sound.soundPlay("card-flip-3")
            self.rev = 3
```



Creating the data table and calculating poker hand strengths

Using Binomial Coefficients to find the combinations

Libraries used:
math

$$\binom{n}{k} = \begin{cases} \frac{n!}{k!\,(n-k)!} & \text{for } 0 \le k < n \\ 0 & \text{otherwise,} \end{cases}$$

```
def combinations(n, k):
    all_possibilities = float(math.factorial(n) / (math.factorial(k) * math.factorial(n - k)))
    return all_possibilities
```

$$\binom{52}{5} = 2,598,960.$$

There are 2.59 million different combinations of 5 card poker hands from a full deck (set of unknown cards)

Then by ranking a hand (and tiebreakers)

```python
def onepair(hand):
    allfaces = [f for f,s in hand]
    allftypes = set(allfaces)
    pairs = [f for f in allftypes if allfaces.count(f) == 2]
    if len(pairs) != 1:
        return False
    allftypes.remove(pairs[0])
    return 'one-pair', pairs + sorted(allftypes,
                                      key=lambda f: face.index(f),
                                      reverse=True)

def highcard(hand):
    allfaces = [f for f,s in hand]
    return 'high-card', sorted(allfaces,
                               key=lambda f: face.index(f),
                               reverse=True)

handrankorder =  (straightflush, fourofakind, fullhouse,
                  flush, straight, threeofakind,
                  twopair, onepair, highcard)

def rank(cards):
    hand = handy(cards)
    for ranker in handrankorder:
        rank = ranker(hand)
        if rank:
            break
    assert rank, "Invalid: Failed to rank cards: %r" % cards
    return rank
```

```python
def pointCalc(hand):
    handy = rank(hand)
    ranker = handy[0]
    tie = handy[1]

    base_value = handpower(ranker)*10000000
    tiepower = 0
    for j,i in enumerate(tie):
        tiepower += (10**j)*rank_compare.power_map[i]
    finalpower = base_value + tiepower
    return finalpower
```

```python
{"royal-flush":0,"straight-flush":1, "four-of-a-kind":2, "full-house":3,
  "flush":4, "straight":5, "three-of-a-kind":6,
  "two-pair":7, "one-pair":8, "high-card":9}
```

```python
# Record the start time
all_start_time = time.perf_counter()

comb_start_time = time.perf_counter()

# Generate all combinations of 5 cards from the sorted deck
card_combinations = list(itertools.combinations(deck, 5))
card_combinations = [sort_hand(comb) for comb in card_combinations]
print(card_combinations[:20])

comb_end_time = time.perf_counter()
comb_elapsed = comb_end_time - comb_start_time
print(f"Time elapsed: {comb_elapsed} seconds")

# Print the total number of combinations
print("Total combinations:", len(card_combinations))
```

```python
cards["rank"] = cards["hand"].apply(collectRank)
print("Ranks done")
cards["tie"] = cards["hand"].apply(collectTie)
print("Ties done in")
cards["points"] = cards["hand"].apply(collectPoint)
print("Ties done in")
print(cards[:20])
cool = cards.sort_values(by="points")
print(cool[:20])

cool.to_csv("cards6.csv",index=False)
```

We can make a data table

Libraries used:
Time
Pandas
itertools

	hand	rank	tie	points
1	2♠ 3♠ 5♠ 4♦ 7♥	high-card	['7', '5', '4', '3', '2']	20023457
2	4♠ 2♣ 7♦ 3♥ 5♥	high-card	['7', '5', '4', '3', '2']	20023457
3	4♠ 2♣ 3♥ 5♥ 7♥	high-card	['7', '5', '4', '3', '2']	20023457
4	5♠ 7♠ 2♣ 4♣ 3♥	high-card	['7', '5', '4', '3', '2']	20023457
5	5♠ 2♣ 4♣ 7♣ 3♥	high-card	['7', '5', '4', '3', '2']	20023457
6	5♠ 2♣ 4♣ 7♦ 3♥	high-card	['7', '5', '4', '3', '2']	20023457
7	5♠ 2♣ 4♣ 3♥ 7♥	high-card	['7', '5', '4', '3', '2']	20023457
8	7♠ 2♣ 4♣ 5♣ 3♥	high-card	['7', '5', '4', '3', '2']	20023457
9	2♣ 4♣ 5♣ 7♣ 3♥	high-card	['7', '5', '4', '3', '2']	20023457
10	2♣ 4♣ 5♣ 7♦ 3♥	high-card	['7', '5', '4', '3', '2']	20023457

That table is then used to go and make hand evaluators to complete game logic

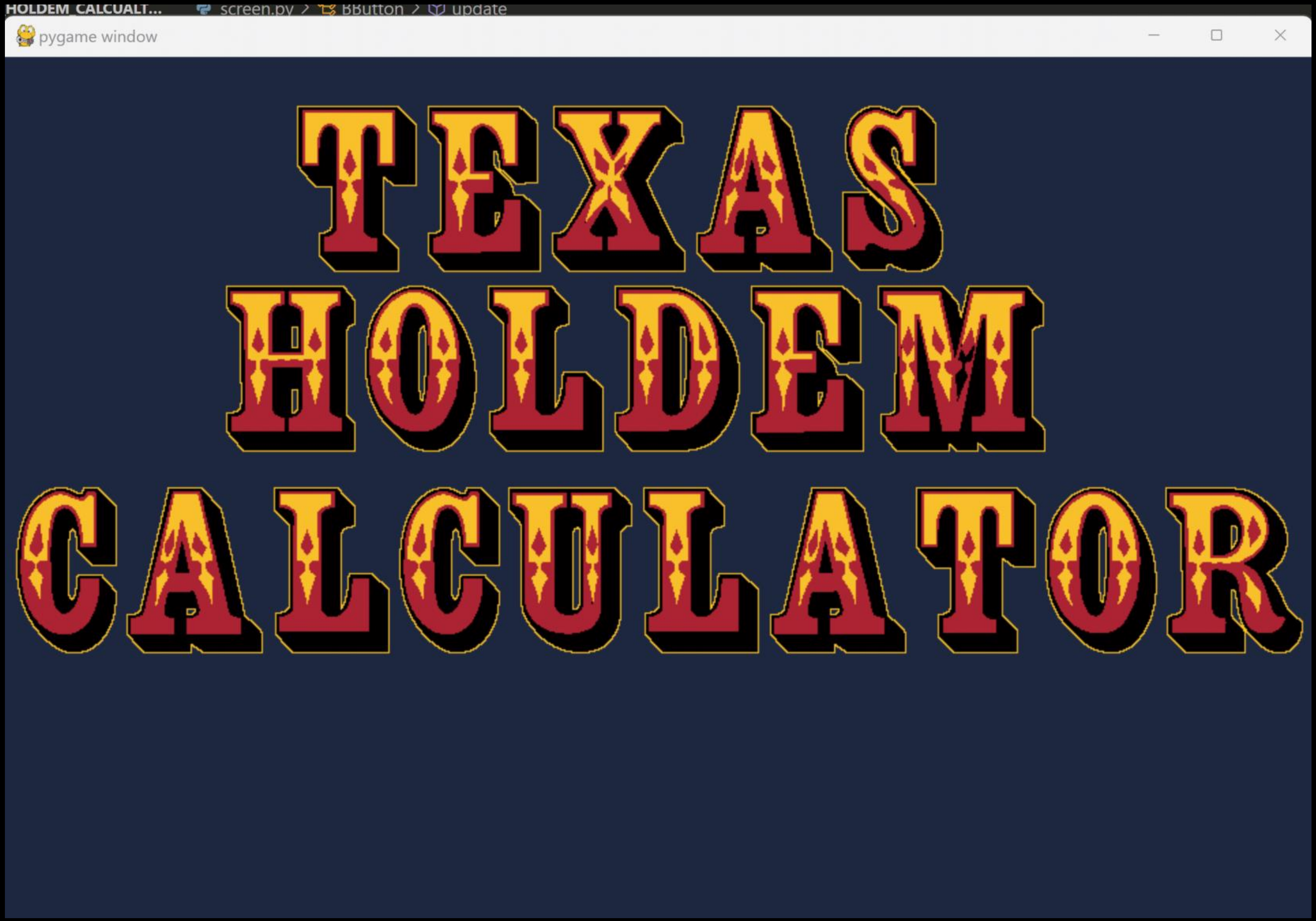
```python
# Function to find the strongest hand among multiple hands using a hand_to_row dictionary
def strongestHand3(hands, hand_to_row):
    rankerhands = []

    for j, i in enumerate(hands):
        # Compress and sort the current hand
        current = compressHand(sort_hand(i))

        # Find the corresponding row using the hand_to_row dictionary for the current hand
        selected_row = hand_to_row[current]
        row_info = selected_row

        # Extract hand-related information from the row dictionary
        handy = row_info["hand"]
        rank = row_info["rank"]
        tie = row_info["points"]
        info = ast.literal_eval(row_info["tie"])
        group = [handy, rank, info, j, tie]
        rankerhands.append(group)

    # Find the strongest hand(s) based on the maximum points value
    max_value = float('-inf')
    max_lists = []

    for i in rankerhands:
        if i[4] > max_value:
            max_value = i[4]
            max_lists = [i]
        elif i[4] == max_value:
            max_lists.append(i)

    # Return the result as a list containing hand, rank, and points
    result = [[i[0], i[1], i[4]] for i in max_lists]
    return result
```

```python
def finalEvaluation(self):
    # Create an empty list for the community hand
    communityHand = []
    self.allStrength = []

    # Convert cards in the community hand to the appropriate format
    for i in self.groups[3].cards:
        communityHand.append(rank_compare.toFormat((i.value, i.suit)))

    # Create an empty list for the player's hand and convert the cards to the appropriate format
    playerHand = []
    for i in self.groups[0].cards:
        playerHand.append(rank_compare.toFormat((i.value, i.suit)))

    # Create a list of enemy hands, converting the cards to the appropriate format
    enemyHands = []
    for j in self.groups[1]:
        temp = []
        for i in j.cards:
            temp.append(rank_compare.toFormat((i.value, i.suit)))
        enemyHands.append(temp + communityHand[:self.rev])

    # Calculate the strength of each enemy hand
    for j, i in enumerate(enemyHands):
        options = rank_compare.gameComp(i)
        if self.points:
            strongest = rank_compare.strongestHand3(options, hand_to_row)
            for i in strongest:
                self.allStrength.append([i[0].split(" "), i[1], i[2], j])

    # Add the player's hand to the list of hands to evaluate
    appender = self.handsToUse[0]
    appender.append("player")
    self.allStrength.append(appender)
```

```python
    # Find the hand(s) with the highest score
    max_value = max(sub_list[2] for sub_list in self.allStrength)
    max_lists = [sub_list for sub_list in self.allStrength if sub_list[2] == max_val

    # Determine the winner and print the results
    if len(max_lists) == 1:
        if not max_lists[0][3] == "player":
            self.groups[1][max_lists[0][3]]
            print("you have been beaten")
            for i in self.allStrength:
                print(f"NO:{i[3]},   Bhand:{i[0]},  type:{i[1]}, score:{i[2]}")
            return self.groups[1][max_lists[0][3]]
        else:
            print("Yay: you won")
            for i in self.allStrength:
                print(f"NO:{i[3]},   Bhand:{i[0]},  type:{i[1]}, score:{i[2]}")
            return self.groups[0]
    else:
        for i in max_lists:
            if i[3] == "player":
                print("It's a draw")
                for i in self.allStrength:
                    print(f"NO:{i[3]},   Bhand:{i[0]},  type:{i[1]}, score:{i[2]}")
                return
            else:
                print("you have been beaten")
                print(f"NO:{i[3]},   Bhand:{i[0]},  type:{i[1]}, score:{i[2]}")
    return
```

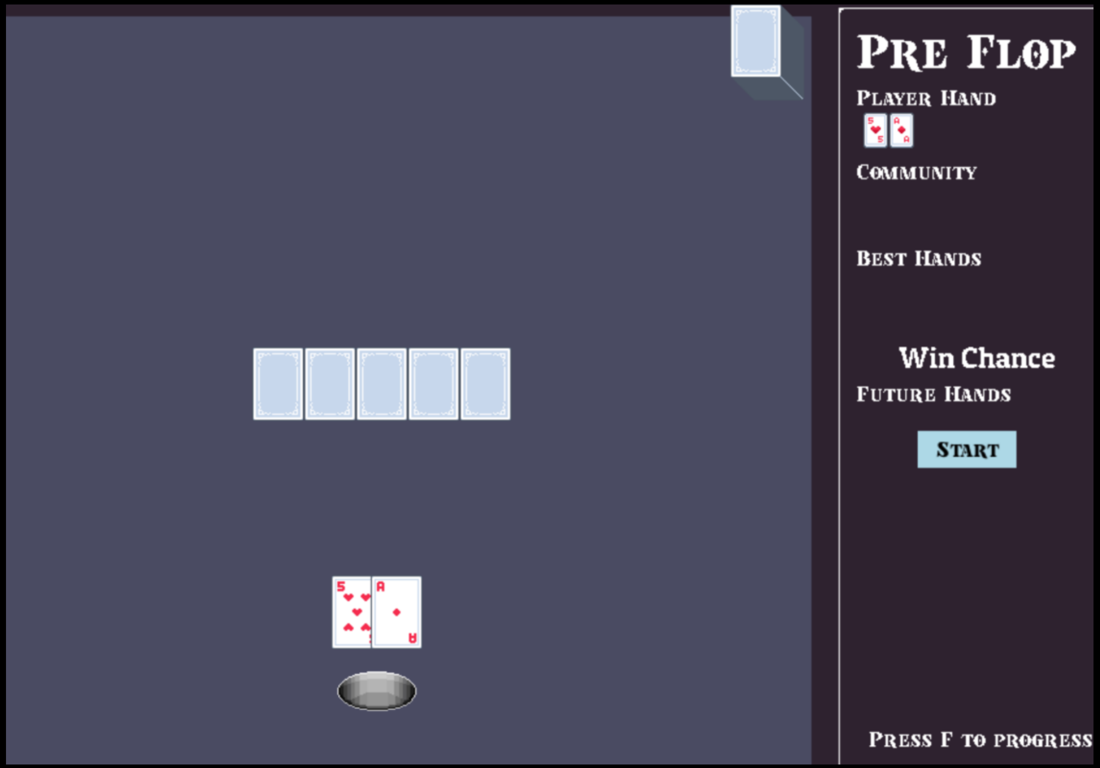
This is all displayed through the GUI

Libraries used:
pygame

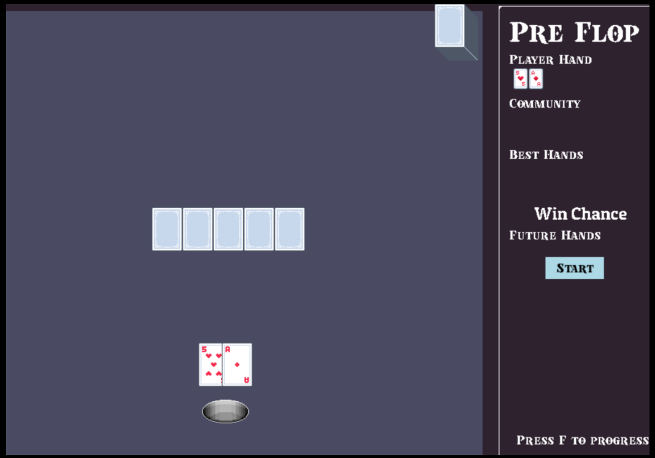


Predicting Future Hands

- In order to actually be able to improve how one might play poker, we need to actually be able to forewarn the players about certain situations?



- We need to look into all of the future combinations that are possible



$\binom{50}{5} \cong$ 2 million future combinations

$\binom{47}{2}$ = 1081 future combinations

$\binom{46}{1}$ = 46 future combinations

Once all the combinations are found, they can all be evaluated with the players hand together.

And then that can be used to find the possibility of each hand rank by how much they occur.

```python
def futurehand(handr, commr, rev=0):
    if gutt == 3:
        left = 2
    if gutt == 4:
        left = 1
    if gutt == 0:
        print("gutter")
        left = 0
    hand = handr
    comm = commr
    comm = comm[:5 - left]
    opti = hand + comm
    decka = [i for i in fullDeck if not i in comm and not i in hand]

    # Generate all possible card combinations for the remaining cards to be drawn
    card_combinations = list(itertools.combinations(decka, left))

    # Combine the current hand and community cards with the possible combinations
    card_combinations = [list(comb) + opti for comb in card_combinations]

    # Calculate the total number of possible combinations
    combine = mt.combinations(len(decka), left)

    # Calculate the strength of each card combination
    allStrength = []
    for j, i in enumerate(card_combinations):
        options = rank_compare.gameComp(i)
        strongest = rank_compare.strongestHand3(options, hand_to_row)
        i = strongest[0]
        allStrength.append([i[0].split(" "), i[1], i[2], j])

    # Print the strengths of all card combinations
    for i in allStrength:
        print(f"NO:{i[3]},   Bhand:{i[0]},  type:{i[1]}, score:{i[2]}")

    # Return the list of strengths and the total number of possible combinations
    return (allStrength, combine)
```

```
♛Best Hands ♛
- royal-flush  was 0 out of 1081.0 | This is a 0.0% chance
- straight-flush  was 0 out of 1081.0 | This is a 0.0% chance
- four-of-a-kind  was 1 out of 1081.0 | This is a 0.09250693802035154% chance
- full-house  was 27 out of 1081.0 | This is a 2.497687326549491% chance
- flush  was 45 out of 1081.0 | This is a 4.162812210915819% chance
- straight  was 0 out of 1081.0 | This is a 0.0% chance
     ♛Total Chance for Best Hand is 6.753006475485662%

♜Solid Hands ♜
- three-of-a-kind  was 63 out of 1081.0 | This is a 5.827937095282146% chance
- two-pair  was 405 out of 1081.0 | This is a 37.46530989824237% chance
     ♜Total Chance for Solid Hand is 43.29324699352451%

♝Regular Hands ♝
- one-pair  was 540 out of 1081.0 | This is a 49.953746530989825% chance
- high-card  was 0 out of 1081.0 | This is a 0.0% chance
     ♝Total Chance for Regular Hand is 49.953746530989825%
total is 1081
  ▨  Minimum standard rank is high-card
  ▨  Total Chance for A Low Probability Hand (<=2%) is 50.046253469010175%
  ▨  Total Chance for A Above Community Hand is 100.0%
```

Simulations

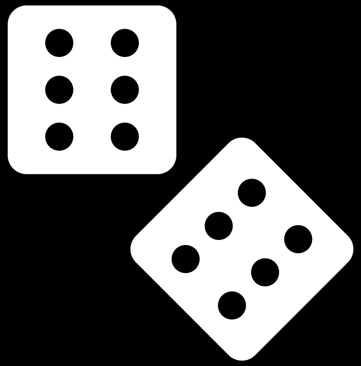
- Monte Carlo is the concept of randomly carrying out an action and measuring it over a certain amount of times to see if there is any consistent value.

- In Pokers case, we want to run as many games as possible in a short amount of time to be able to figure out facts about certain game states

- Like if changing the amount of players increases or decreases your chance of winning

- Or if a certain hand is better than others to start with

- We can do this by optimising our current code to run really fast



Here, a lot of the code for the GUI has been removed. Instead, we have a very compact version of the game that can run a large amount of games fairly fast

```python
def sim(sim=1, players=2, speed="instant", inject=False, phase=1, hand=['8♠', 'k♠'], comm=['a♠', '8♣', 'a♣', 'a♥', '2♥']):
    # Set the game phase to 0
    game.phase = 0

    # Initialize empty hands for each player
    for i in range(players):
        temphand.append(EHand())

    # Set up the game groups (main hand, enemy hands, deck, and community cards)
    game.groups = (mainhand, temphand, deck, commune)

    # Initialize win, loss, and draw counters
    Counter.wins = 0
    Counter.losses = 0
    Counter.draws = 0

    # Start a timer to measure the simulation time
    rank_start_time = time.perf_counter()

    # If the 'inject' option is enabled, set up the game with the provided hand, community cards, and phase
    if inject:
        Counter.player = hand
        Counter.community = comm
        Counter.phase = phase
        for i in range(sim):
            simulateGame3(i)

    # Run the simulation with the specified speed
    if speed == "instant":
        for i in range(sim):
            simulateGame2(i)
    else:
        for i in range(sim):
            simulateGame(i)

    # Calculate and print the time elapsed during the simulation
    rank_end_time = time.perf_counter()
    rank_elapsed = rank_end_time - rank_start_time
    print(f"Time elapsed: {rank_elapsed} seconds")

    # Print the number of wins and losses
    print(f"Wins: {Counter.wins} Losses: {Counter.losses}")
```

```
instant or looped: instant
how many simulations: 1000
how many opponents: 5
Do you have a state to load up?
Time elapsed: 2.2463131999975303 seconds
Wins: 159 Losses: 815 Draws: 26
PS C:\DevWork\UNI\MKU_Assignment-Texas_Holdem_Calcualtor->
```

```python
#plays a round of poker instantly
def round(self):
    self.draw(*self.groups)
    self.rev = 5
    self.grabEvaluation()
    self.finalEvaluation()
    self.cleanup(*self.groups)
```

Insights

Keep in mind…

- When predicting, we are assuming that **no one is folding**.
  - This means that you could be losing to someone who could've folded in a real world situation.
  - This pretty much makes any win loss ratio inaccurate to the real thing
- So all results from the simulations as well as have to be taken with a grain of salt.

```python
    def simulateSample():
        numro = 0
        rank_start_time = time.perf_counter()
        for countsim in simAmmount:
            Counter.resetCounter()
            for h,i in enumerate(card_combinations): #1326
                for j in enemyAmount:
                    numro += 1
                    Counter.current = list(i)
                    sim(sim=countsim,players=j,speed="instant",inject=True,phase=1,hand=list(i),comm=['a♠', '8♣',
                if h % (countYakno//100) == 0:
                    print(h,(countYakno//100))
                    curren = time.perf_counter()
                    rank_elapsed = curren - rank_start_time
                    percental += 1
                    secs,mins = math.modf(rank_elapsed/60)
                    print(f"Creation {percental}% complete - Current Time Elapsed {int(mins)}:{round(secs*60,2)} '
                #print(numro)
            print(f"Creation 100% complete")
            print("Generating files...")
            Counter.generateFile()

    rank_start_time = time.perf_counter()
    simulateSample()
```

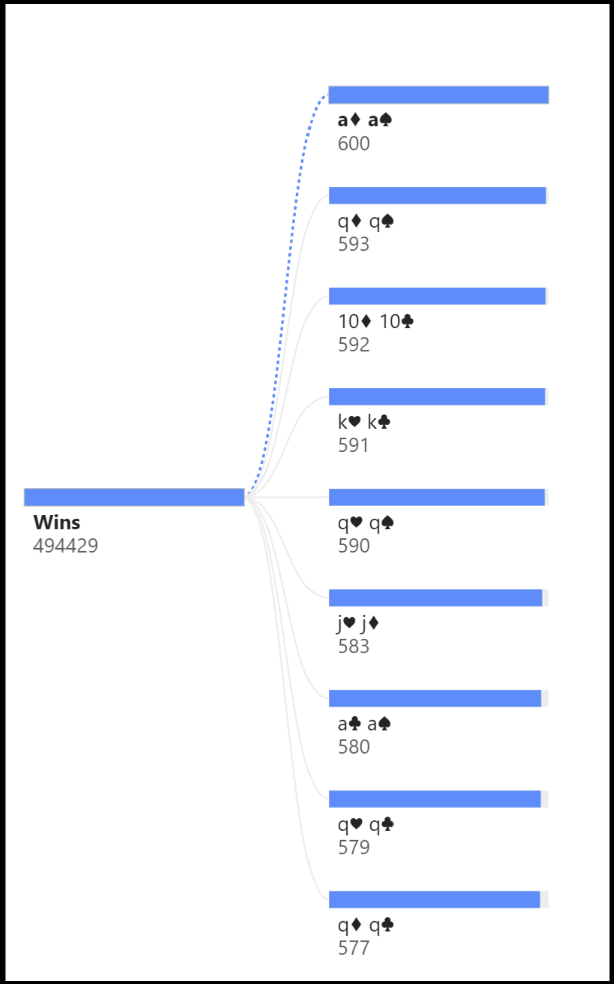
Here we are utilising the Monte Carlo to take information about each game being simulated.

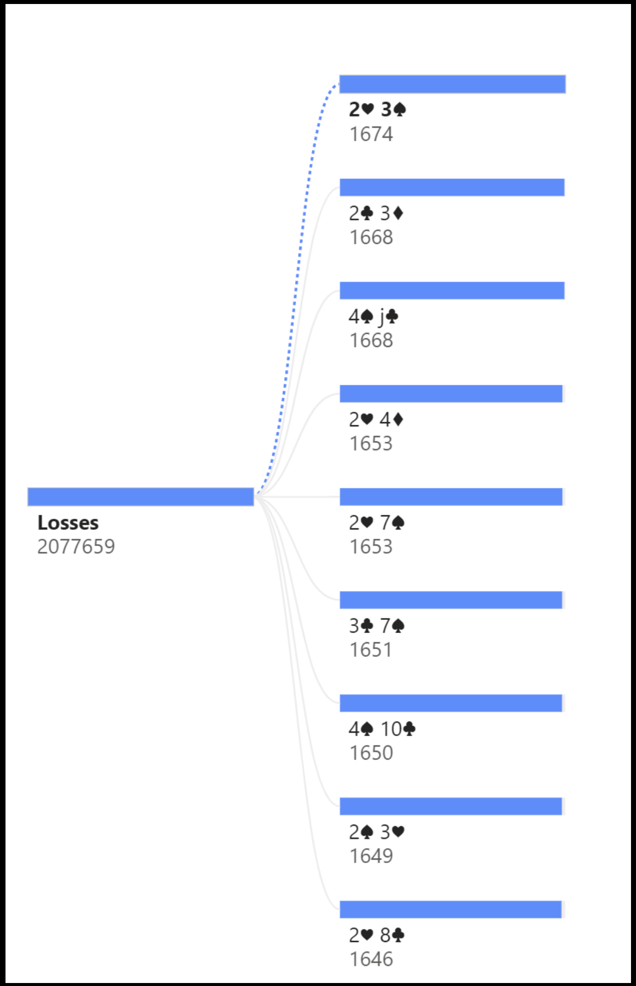
```python
@staticmethod
def generateFile():
    table = {"hand":Counter.hand,"outcome":Counter.outcome,"popular freq":Counter.freqc,"freq volume":Counter.freqa,"best":Counter.best,"opponents":
    tableex = pd.DataFrame(table)
    str(Counter.wins*Counter.losses*Counter.draws)
    tableex.to_csv(str((Counter.wins*Counter.losses*Counter.draws)/100)+str(np.random.randint(1,100))+"sim with"+str(Counter.players)+"players.csv")
```

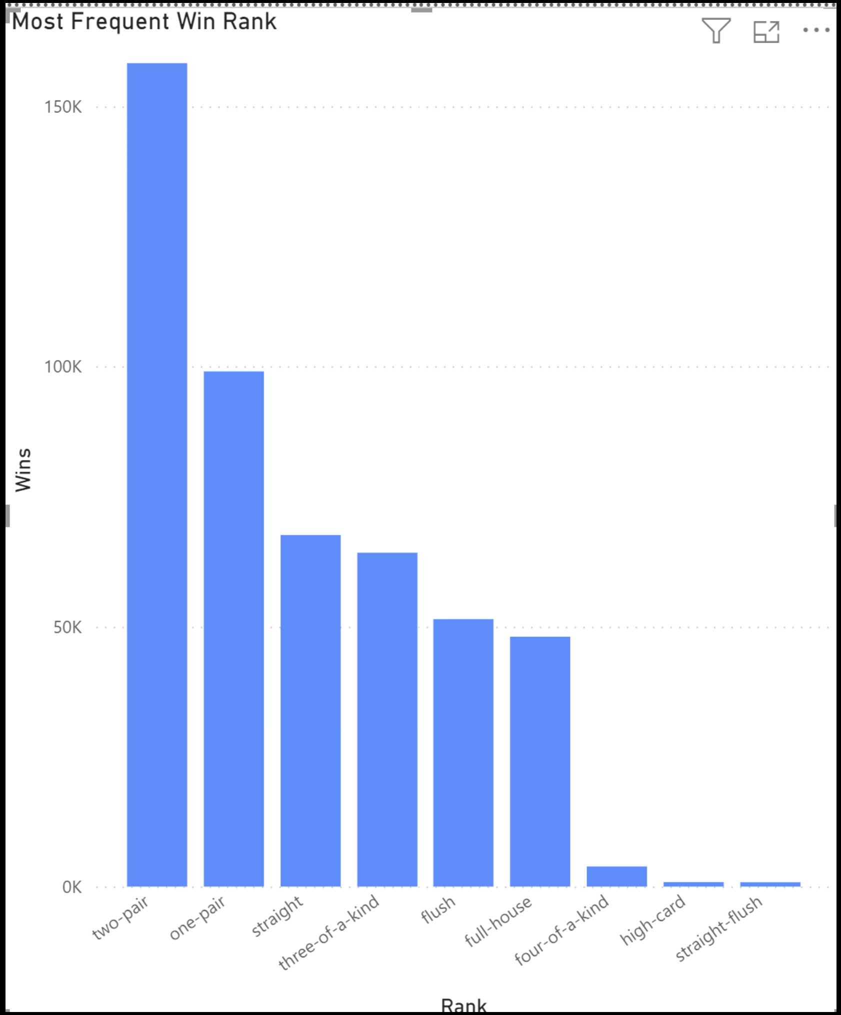
		hand	outcome	popular freq	freq volume	best	opponents
1	0	2♥ 2♦	win	high-card	3	three-of-a-kind	4
2	1	2♥ 2♦	loss	one-pair	3	two-pair	4
3	2	2♥ 2♦	loss	one-pair	3	straight	4
4	3	2♥ 2♦	loss	one-pair	4	straight	4
5	4	2♥ 2♦	loss	two-pair	4	full-house	4
6	5	2♥ 2♦	draw	two-pair	6	two-pair	4
7	6	2♥ 2♦	loss	one-pair	3	one-pair	4
8	7	2♥ 2♦	loss	one-pair	4	straight	4
9	8	2♥ 2♦	loss	full-house	4	four-of-a-kind	4
10	9	2♥ 2♦	loss	one-pair	3	flush	4
11	10	2♥ 2♦	win	three-of-a-kind	4	full-house	4
12	11	2♥ 2♦	loss	two-pair	3	full-house	4
13	12	2♥ 2♦	win	two-pair	3	full-house	4
14	13	2♥ 2♦	loss	high-card	1	flush	4
15	14	2♥ 2♦	loss	two-pair	3	full-house	4

The dataset has the following:

- hand: The specific hole cards dealt to the player (e.g., 2♥ 2♦).
- outcome: Whether the player won or lost the hand (e.g., win).
- popular freq: The most common hand ranking achieved by all players (e.g., high-card).
- freq volume: The number of times the most common hand ranking occurred (e.g., 3).
- best: The best hand ranking the player managed to achieve with that specific hand (e.g., three-of-a-kind).
- opponents: The number of opponents the player faced in each game (e.g., 4).





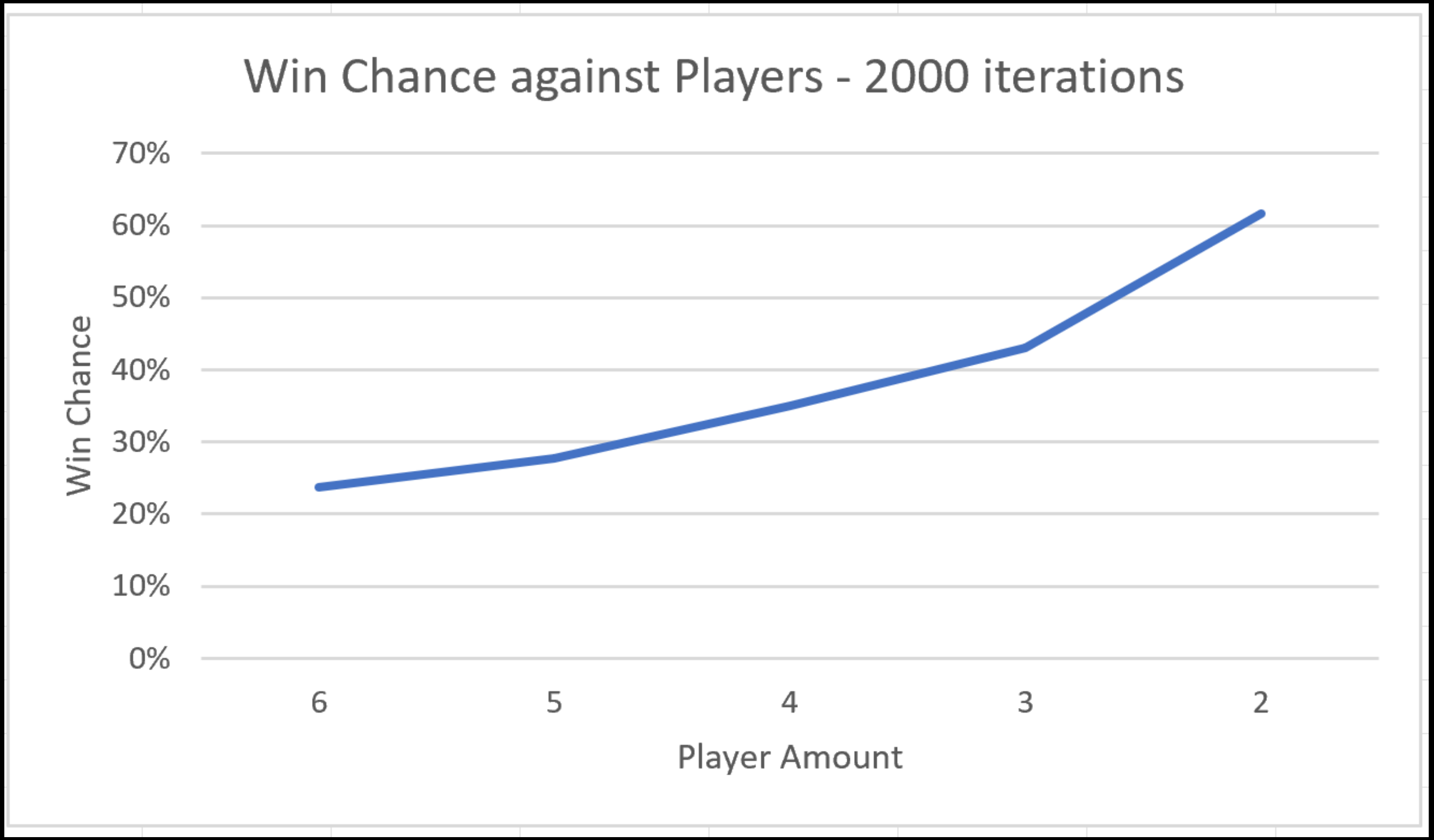


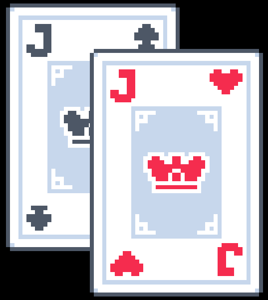
Using the dataset, we found the top cards for 2000 iterations of games that won.

We can also see the cards that lost the most

We can also see what type of hand wins the most

When changing the amount of players for a strong hand , we found that the less players there was, the more likely you were to win.





Future Implementations

What other ways could we have done insight? Could you add more than one deck?, Betting and Folding?

Reference

Wolfram Research. (2021). Binomial coefficient. MathWorld. https://mathworld.wolfram.com/BinomialCoefficient.html

Brilliant. (n.d.). Math of poker. Brilliant.org. Retrieved from https://brilliant.org/wiki/math-of-poker/

Code For the Project. (2023). https://github.com/LevianDanProduction/MKU_Assignment-Texas_Holdem_Calculator

Questions?