Application of Combinatorics in Big Two

Bob Kunanda - 13523086
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*bobkunanda@gmail.com*, *13523086@std.stei.itb.ac.id*

***Abstract*— This paper explores the use of combinatorics and Monte Carlo simulations to analyze and calculate the probabilities of various card combinations in a hand. Combinatorics provides precise mathematical equations for simple cases, while Monte Carlo simulations are employed to address the complexity of larger, more intricate scenarios, such as calculating the likelihood of straights, flushes, full houses, and other hands in a deck. The paper also introduces practical implementations of these models, with code designed to compute results efficiently. By combining theoretical foundations with computational methods, the work aims to offer tools and insights that can be extended for applications like AI model development for the strategic card game Big Two. The equations and Python code presented provide a foundational framework that balances accuracy and efficiency, making them adaptable for both academic and practical uses.**

Keywords—Big Two, card combinations, combinatorics, Monte Carlo simulation

I. Introduction

Big Two (also known as Capsa or Cus in Indonesia) is a shedding-type card game of Cantonese origin. The game is popular in Asia, and it is played by 2 to 4 players with a 52-card deck. The goal of the game is to be the first to play all the cards by forming valid combinations, such as singles, pairs, triple, or poker-style hands.

Each player is given 13 random sets from a deck of shuffled cards. In Big Two, cards are ranked both by their values and suits. The ranking order is as follows value wise from lowest to highest would be 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A, 2 and suits wise it would be Diamonds (♦), Clubs (♣), Hearts (♥), Spades (♠). There are 4 valid combinations: singles or any single card, pairs or two cards of the same rank, triple, and five-card poker hands like straight, flush, full house, four-of-a-kind (plus one single), or straight flush.

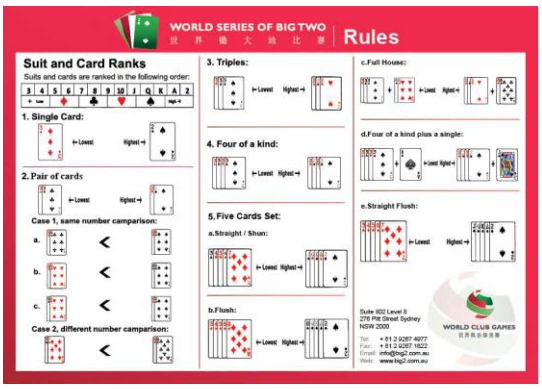


Image 1. Rules of Big Two, taken from [1]

The game starts with whoever got the smallest value card (3♦), then each subsequent player must play a higher value card or combination than the one before with the same number of cards. When all but one player passed in succession, the round is over and used card are put in the waste pile. Then the next round starts with the last player playing.

II. Theoretical Foundation

A. Combinatorics

Combinatorics is a branch of mathematics that studies the counting, arrangement, and combination of objects. It focuses on finding efficient methods to count and organize discrete structures without the need to enumerate all possible configurations. Combinatorics has applications in many fields, including computer science, cryptography, and statistical physics. It provides tools for solving problems related to counting, probability, and the arrangement of sets of objects.

B. Principle of Inclusion-Exclusion

Combinatorics is a branch of mathematics that studies the counting, arrangement, and combination of objects. It focuses on finding efficient methods to count and organize discrete structures without the need to enumerate all possible configurations. Combinatorics has applications in many fields, including computer science, cryptography, and statistical physics. It provides tools for solving problems related to counting, probability, and the arrangement of sets of objects.

For any finite sets $A_1, A_2, \cdots, A_n$:

$$| A1 \cup A2 \cup \cdots \cup An | \sum_{i=1}^{n} |A_i| - \sum_{1 \le i < j \le n} |A_i \cap A_j| + \sum_{1 \le i < j < k \le n} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{n+1} |A_1 \cap A_2 \cap \cdots \cap A_n|$$

C. Permutation

Permutations are a fundamental concept in combinatorics, referring to the arrangement of objects in a specific order. The study of permutations helps in understanding how different sequences can be formed from a given set of elements. Permutations are widely used in various fields, including mathematics, computer science, and operations research.

A permutation of a set is an arrangement of its elements in a specific sequence or order. If we have a set of n distinct elements, the number of possible permutations of these elements is denoted as $n!$ (n factorial), which is the product of all positive integers up to

$$n: n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

C. Combination

Combinations are a key concept in combinatorics, referring to the selection of items from a larger set where the order of selection does not matter. The study of combinations helps in understanding how different groups can be formed from a given set of elements. Combinations are widely used in various fields, including mathematics, statistics, and computer science.

A combination of a set is a selection of its elements without regard to the order in which they are selected. If we have a set of n distinct elements, and we want to choose r elements from this set, the number of possible combinations is denoted as $\left(\frac{n}{r}\right)$ (read as "n choose r") and is given by the formula: $\frac{n}{r} = \frac{n!}{r!(n-r)!}$ where $n!$ (n factorial) is the product of all positive integers up to n.

D. Monte-Carlo simulation

Calculating probabilities in complex card games like determining the likelihood of specific outcomes is challenging because of the large number of possible hands, overlapping patterns, and interdependence among cards. The combinatorial nature of such problems often involves evaluating subsets, accounting for overlaps, and ensuring no double-counting, which becomes computationally intractable as the number of cards increases. This huge and complex game requires a simulation to identify the probability.

III. METHODOLOGY

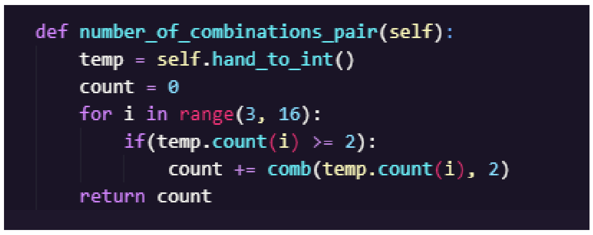
To simplify equation the total number of cards in a hand will be written as $total_{value(card\ value)}$and total number of card of a certain suit in a hand will be written as $total_{suit(card\ suit)}$. To simplify the equations, we will introduce specific notations for better clarity and manageability. The total number of cards of a particular rank or value in a player's hand will be denoted as $total_{value(card\ value)\_}$, where *card value* represents the specific rank (e.g., Ace, 2, 3, etc.). Similarly, the total number of cards of a particular suit in a player's hand will be expressed as $total_{suit(card\ suit)}$, where *card suit* corresponds to one of the four suits (e.g., Hearts, Diamonds, Clubs, Spades). These notations allow us to succinctly represent counts in a player's hand for both specific values and suits, making it easier to construct equations and analyze scenarios systematically.

A. Combinations of Pairs

To define the combination of pairs in a single hand of 13 cards, we consider the range of card values, with 3 being the smallest value card and 2 being the largest value card (often referred to as the "Big Two" card game ranking). The formula is constructed to iterate through all possible pairs of cards, symbolizing the combination of cards that can form pairs.

$$total_{pair} = \sum_{i=3}^{2} C \binom{total_{value(i)}}{2}$$

The number of combinations of a pair equation, when implemented in Python, would look something like this: hands_to_int is a function that converts the hand from its original values (e.g., rank and suit representations) into an array of integers representing the hand. This array facilitates easier mathematical operations and comparisons when calculating combinations. The function typically uses a mapping of card ranks (e.g., { '3': 3, ..., '2': 15}) to integers and processes each card in the hand to replace its rank with the corresponding integer value. This makes it possible to evaluate combinations efficiently, especially when paired with libraries like math for generating pairs, triples, or other subsets from the hand.



```
def number_of_combinations_pair(self):
    temp = self.hand_to_int()
    count = 0
    for i in range(3, 16):
        if(temp.count(i) >= 2):
            count += comb(temp.count(i), 2)
    return count
```

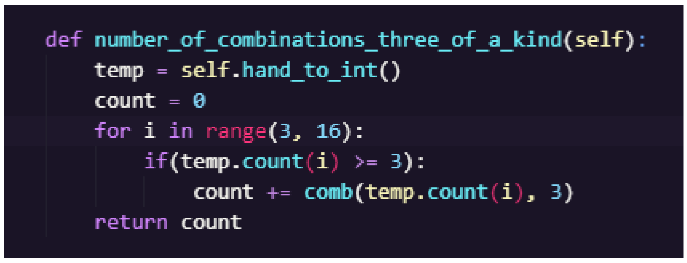
Image 2. Number of combinations of pairs code, taken from [2]

B. Combinations of threes

Like combinations of pairs, the combinations of threes can be defined using a similar approach but with modified conditions and combination parameters. Specifically, the total number of triples can be calculated using the formula:

$$total_{triple} = \sum_{i=3}^{2} C \binom{total_{value(i)}}{3}$$

Combinations for threes just takes the program from combinations of pairs and change the condition and the combination parameters.



```
def number_of_combinations_three_of_a_kind(self):
    temp = self.hand_to_int()
    count = 0
    for i in range(3, 16):
        if(temp.count(i) >= 3):
            count += comb(temp.count(i), 3)
    return count
```

Image 3. Number of combinations of triple code, taken from [2]

C. Combinations of Straight

To calculate the combinations of straights in two different cases, no separate straight and separate straight, the logic becomes increasingly complex as we consider both overlapping and non-overlapping sequences. Here's how the explanation can be expanded for clarity

1. No Separate Straight

   The case of no separate straight refers to the scenario where only a single straight exists, and it consists of more than five consecutive cards. In this situation, the straight is contiguous, with all its values forming a single sequence without any gaps or interruptions. The equation for this is:

$$total_{straight} = If\ (value_i = value_{i+1} - 1 = \cdots = value_{i+4} - 4),\ \sum_{i=start}^{finish-4} \prod_{k=0}^{4} total_{value(i+k)}$$

Here's a breakdown of the terms:

- $value_i$: Represents the rank of the i-th card.
- $start$: The rank of the first card in the straight.
- $finish$: The rank of the last card in the straight.
- $\prod_{k=0}^{4} total_{value(i+k)}$ Computes the total combinations by multiplying the counts of cards for each consecutive rank forming the straight.

This ensures that only one continuous straight is counted, avoiding any separation or gaps. The condition $(value_i = value_{i+1} - 1 = \cdots = value_{i+4} - 4)$enforces the sequence.

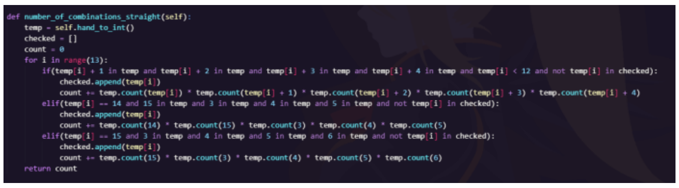
2. Separate Straight

   The case of separate straight refers to the presence of two independent straights that are not connected or overlapping. This scenario involves a more complex calculation, as it requires identifying two distinct groups of consecutive cards within the hand. The equation for this is:

$$total_{straight} = If\ (value_i = value_{i+1} - 1 = \cdots = value_{i+4} - 4)\ \sum_{i=start1}^{finish1-4} \prod_{k=0}^{4} total_{value(i+k)} + \sum_{j=start2}^{finish2-4} \prod_{m=0}^{4} total_{value(j+m)}$$

   The equation essentially iterates over all possible positions for the two separate straights, ensuring no overlap, and multiplies the combinations for each straight.

The number of combinations of straights can be simplified in implementation to make the code more readable and maintainable. While the mathematical approach might be detailed and complex, simplifying the logic and focusing on clear iteration and conditions will produce the same results without altering correctness.



```python
def number_of_combinations_straight(self):
    temp = self.hand_to_int()
    checked = []
    count = 0
    for i in range(13):
        if(temp[i] + 1 in temp and temp[i] + 2 in temp and temp[i] + 3 in temp and temp[i] + 4 in temp and temp[i] < 12 and not temp[i] in checked):
            checked.append(temp[i])
            count += temp.count(temp[i]) * temp.count(temp[i] + 1) * temp.count(temp[i] + 2) * temp.count(temp[i] + 3) * temp.count(temp[i] + 4)
        elif(temp[i] == 14 and 15 in temp and 3 in temp and 4 in temp and 5 in temp and not temp[i] in checked):
            checked.append(temp[i])
            count += temp.count(14) * temp.count(15) * temp.count(3) * temp.count(4) * temp.count(5)
        elif(temp[i] == 15 and 3 in temp and 4 in temp and 5 in temp and 6 in temp and not temp[i] in checked):
            checked.append(temp[i])
            count += temp.count(15) * temp.count(3) * temp.count(4) * temp.count(5) * temp.count(6)
    return count
```

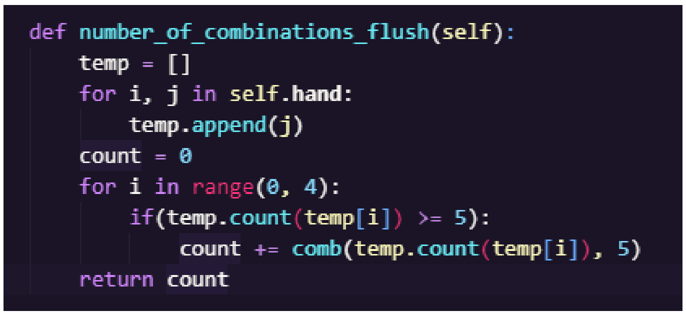
Image 4. Number of combinations of straight code, taken from [2]

D. Combinations of Flush

To calculate the combinations of flushes, the formula involves determining the total number of ways to select 5 cards from each suit and summing them up. The equation can be written as:

$$total_{flush} = C\binom{total_{suit(diamond)}}{5} + C\binom{total_{suit(clover)}}{5} + C\binom{total_{suit(heart)}}{5} + C\binom{total_{suit(spade)}}{5}\mathrm{T}$$

The number of combinations of flushes puts each number of cards from each suit to an array to be processed further.



```python
def number_of_combinations_flush(self):
    temp = []
    for i, j in self.hand:
        temp.append(j)
    count = 0
    for i in range(0, 4):
        if(temp.count(temp[i]) >= 5):
            count += comb(temp.count(temp[i]), 5)
    return count
```

Image 5. Number of combinations of flush code, taken from [2]

E. Combinations of Full House

This equation uses $count_{pair}$, $count_{triple}$, and $count_{fours}$ to symbolize the number of pairs, triples, and fours in a deck. to determine all possible full houses. The unsimplified version breaks down each component of the calculation for clarity and readability, even though it can be expressed in a more compact form.

$$total_{full\ house} = \left(C\binom{4}{2} \times C\binom{3}{3} + C\binom{4}{3} \times C\binom{3}{2}\right) \times count_{triple} \times count_{fours} + C\binom{4}{3} \times C\binom{4}{2} \times P\binom{count_{fours}}{2} + C\binom{3}{3} \times C\binom{3}{2} \times P\binom{count_{triple}}{2} + \left(C\binom{4}{3} \times C\binom{2}{2} \times count_{fours} + C\binom{3}{3} \times C\binom{2}{2} \times count_{triple}\right) \times count_{pair}$$

Breakdown of the Terms

First Term:

This part accounts for full houses formed by a triple from a three-of-a-kind and a pair from a four-of-a-kind. $C\binom{4}{2}$: Ways to choose 2 cards from a four-of-a-kind to form a pair. $C\binom{3}{3}$ ways to choose all 3 cards from a triple to form the three-of-a-kind. As well as full houses formed by a pair from a three-of-a-kind and a triple from a four-of-a-kind. $C\binom{4}{3}$: Ways to choose 3 cards from a four-of-a-kind to form a three-of-a-kind. $C\binom{3}{2}$ ways to choose all 2 cards from a triple to form pair. $count_{triple} \times count_{fours}$: Total combinations for selecting these cards.

Second Term:

Accounts for full houses are formed by selecting a pair and a triple from two different four-of-a-kinds.

$C\binom{4}{3} \times C\binom{4}{2} \times P\binom{count_{fours}}{2}$: Permutations of two distinct four-of-a-kinds.

Third Term:

Accounts for full houses are formed by selecting pairs and triples from two different three-of-a-kinds.

$C\binom{3}{3} \times C\binom{3}{2} \times P\binom{count_{triple}}{2}$: Permutations of two distinct three-of-a-kinds.

Fourth Term:

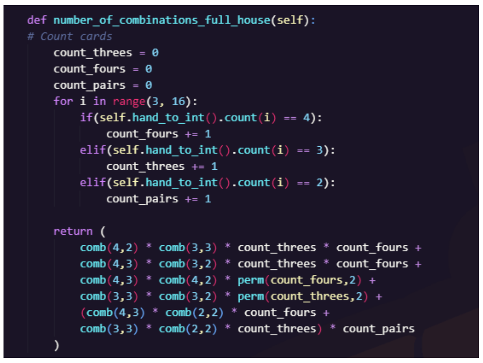
Combines cases where pairs are formed using two cards from a pair and triples from a three-of-a-kind or four-of-a-kind.

- $C\binom{4}{3}$: Ways to choose 3 cards from a four-of-a-kind.
- $C\binom{3}{3}$: Ways to choose 3 cards from a triple.
- $C\binom{2}{2}$: Ways to form a pair from an existing pair.
- $count_{fours}$ and $count_{triple}$: Counts of four-of-a-kinds and three-of-a-kinds.

The simplified version aggregates the terms with precomputed coefficients for clarity:

$$total_{fullhouse} = 18count_{triple} \times count_{fours} + 24P\binom{count_{fours}}{2} + 3P\binom{count_{triple}}{2} + (4count_{fours} + count_{triple}) \times count_{pair}$$

This code for number of combinations of full house used the unsimplified version of the equation to make it readable.



```python
def number_of_combinations_full_house(self):
# Count cards
    count_threes = 0
    count_fours = 0
    count_pairs = 0
    for i in range(3, 16):
        if(self.hand_to_int().count(i) == 4):
            count_fours += 1
        elif(self.hand_to_int().count(i) == 3):
            count_threes += 1
        elif(self.hand_to_int().count(i) == 2):
            count_pairs += 1

    return (
        comb(4,2) * comb(3,3) * count_threes * count_fours +
        comb(4,3) * comb(3,2) * count_threes * count_fours +
        comb(4,3) * comb(4,2) * perm(count_fours,2) +
        comb(3,3) * comb(3,2) * perm(count_threes,2) +
        (comb(4,3) * comb(2,2) * count_fours +
        comb(3,3) * comb(2,2) * count_threes) * count_pairs
    )
```

Image 6. Number of combinations of full house code, taken from [2]

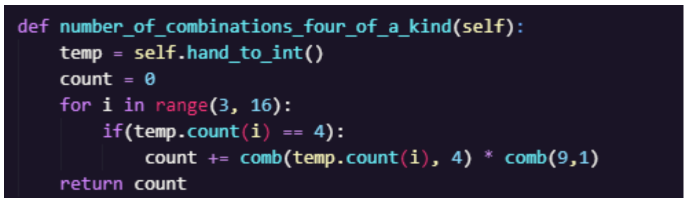
F. Combinations of Four of a Kind

The combination of four of a kind by using an equation is as follows.

$$total_{four\ of\ a\ kind} = \sum_{i=3}^{2} C\binom{total_{value(i)}}{4} \times C\binom{9}{1}$$

- $C\binom{total_{value(i)}}{4}$: The number of ways to select all 4 cards of a single rank.
- $C\binom{9}{1}$): The number of ways to select 1 additional card from the remaining 9 ranks (to make the total hand size 5 cards).
- $\sum_{i=3}^{2}$: Iterates over all ranks to calculate this for each possible four of a kind in the deck.

The code implementation of number of combinations of four of a kind is as follows.



```python
def number_of_combinations_four_of_a_kind(self):
    temp = self.hand_to_int()
    count = 0
    for i in range(3, 16):
        if(temp.count(i) == 4):
            count += comb(temp.count(i), 4) * comb(9,1)
    return count
```

Image 7. Number of combinations of four of a kind code, taken from [2]

G. Combinations of Straight Flush

The formula for calculating the total number of straight flushes considers sequences of cards within each suit that form a consecutive straight. A straight flush requires 5 or more cards of the same suit in consecutive ranks. To account for all possible straight flushes in a deck, the calculation needs to evaluate each suit independently and consider all valid ranges of consecutive cards.

$$\begin{aligned} total_{straight\ flush} &= If\ (value_i = value_{i+1} - 1 = \cdots \\ &= value_{i+4} - 4), \\ &finish_{diamond} - start_{diamond} - 3 + \\ &finish_{clover} - start_{clover} - 3 + \\ &finish_{heart} - start_{heart} - 3 + \\ &finish_{spade} - start_{spade} - 3 \end{aligned}$$

Explanation of Terms

Straight Flush Condition:

- The condition $value_i = value_{i+1} - 1 = \cdots = value_{i+4} - 4$ ensures that the cards form a valid sequence of 5 or more consecutive values.

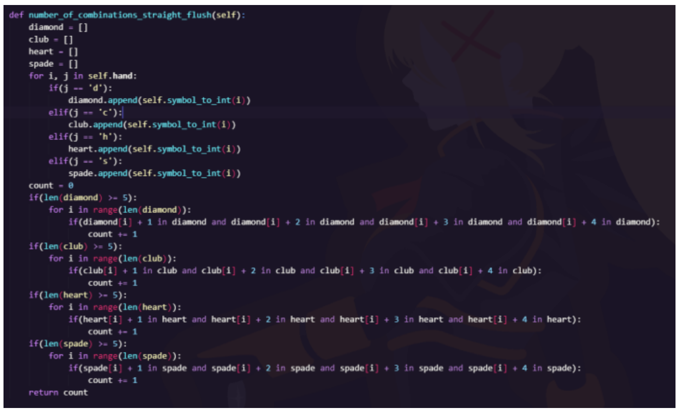
Range Calculation:

- Each suit (diamonds, clubs, hearts, spades) is evaluated independently.
- $start_{suit}$: The rank of the first card in the sequence for the specific suit.
- $finish_{suit}$: The rank of the last card in the sequence for the specific suit.
- $finish_{suit} - start_{suit} - 3$: This calculates the number of valid straight flushes within the suit. Subtracting 3 accounts for the fact that the minimum sequence length is 5 cards.

Summing Across Suits:

- Each suit's valid straight flush combinations are summed to get the total number of straight flushes in the deck.

The code implementation for calculating the total number of straight flush combinations is as follows. The logic has been slightly adjusted for better readability and maintainability while ensuring that the results remain consistent with the original approach. This version focuses on clarity, making it easier to understand the process of iterating through each suit, checking for valid sequences, and calculating the total number of straight flushes across all suits. Below is the Python code that efficiently computes the combinations without compromising on accuracy.



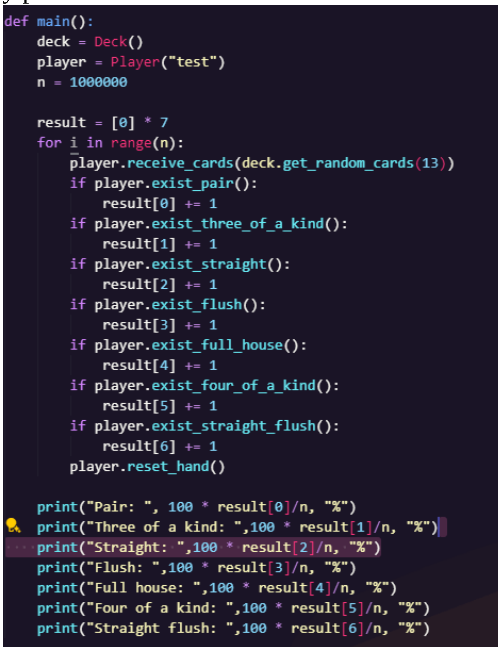
```python
def number_of_combinations_straight_flush(self):
    diamond = []
    club = []
    heart = []
    spade = []
    for i, j in self.hand:
        if(j == 'd'):
            diamond.append(self.symbol_to_int(i))
        elif(j == 'c'):
            club.append(self.symbol_to_int(i))
        elif(j == 'h'):
            heart.append(self.symbol_to_int(i))
        elif(j == 's'):
            spade.append(self.symbol_to_int(i))
    count = 0
    if(len(diamond) >= 5):
        for i in range(len(diamond)):
            if(diamond[i] + 1 in diamond and diamond[i] + 2 in diamond and diamond[i] + 3 in diamond and diamond[i] + 4 in diamond):
                count += 1
    if(len(club) >= 5):
        for i in range(len(club)):
            if(club[i] + 1 in club and club[i] + 2 in club and club[i] + 3 in club and club[i] + 4 in club):
                count += 1
    if(len(heart) >= 5):
        for i in range(len(heart)):
            if(heart[i] + 1 in heart and heart[i] + 2 in heart and heart[i] + 3 in heart and heart[i] + 4 in heart):
                count += 1
    if(len(spade) >= 5):
        for i in range(len(spade)):
            if(spade[i] + 1 in spade and spade[i] + 2 in spade and spade[i] + 3 in spade and spade[i] + 4 in spade):
                count += 1
    return count
```

Image 8. Number of combinations of straight flush code, taken from [2]

G. Monte-Carlo Simulation

Because of the complexity involved in counting all possible outcomes in a deck of cards, a Monte Carlo simulation offers a practical solution to estimate probabilities. This approach uses random sampling to approximate results, by passing the need for exhaustive enumeration of all combinations. Although the results from simulation are not exact, they become increasingly accurate as the number of iterations grows. By running the simulation on a large scale, such as one million randomized hands, the differences between simulated and exact results remain minimal, providing a reliable estimate.

The simulation is implemented using the random library in Python to generate hands and calculate the desired outcomes. By efficiently managing randomness and focusing on specific conditions, the code ensures clarity while maintaining performance. With this design, it can be easily extended or modified to suit different card games or custom deck configurations. Below is the Python code, demonstrating the use of Monte Carlo simulation for solving complex card probability problems.



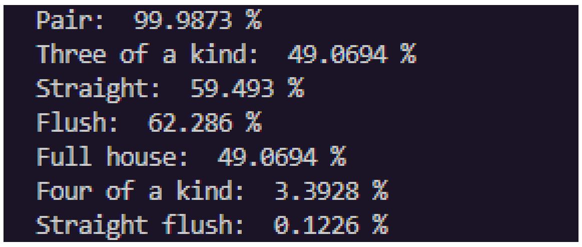
```python
def main():
    deck = Deck()
    player = Player("test")
    n = 1000000

    result = [0] * 7
    for i in range(n):
        player.receive_cards(deck.get_random_cards(13))
        if player.exist_pair():
            result[0] += 1
        if player.exist_three_of_a_kind():
            result[1] += 1
        if player.exist_straight():
            result[2] += 1
        if player.exist_flush():
            result[3] += 1
        if player.exist_full_house():
            result[4] += 1
        if player.exist_four_of_a_kind():
            result[5] += 1
        if player.exist_straight_flush():
            result[6] += 1
        player.reset_hand()

    print("Pair: ", 100 * result[0]/n, "%")
    print("Three of a kind: ",100 * result[1]/n, "%")
    print("Straight: ",100 * result[2]/n, "%")
    print("Flush: ",100 * result[3]/n, "%")
    print("Full house: ",100 * result[4]/n, "%")
    print("Four of a kind: ",100 * result[5]/n, "%")
    print("Straight flush: ",100 * result[6]/n, "%")
```

Image 9. Monte-Carlo simulation code, taken from [2]

The results are stored in a `result` array, where each index corresponds to the count of a specific hand type observed during the simulation. The mapping is as follows:

- `result[0]`: Number of pair occurrences
- `result[1]`: Number of triple occurrences
- `result[2]`: Number of straight occurrences
- `result[3]`: Number of flush occurrences
- `result[4]`: Number of full house occurrences
- `result[5]`: Number of four-of-a-kind occurrences
- `result[6]`: Number of straight flush occurrences

Here is one of the test results from running the simulation:



```
Pair:  99.9873 %
Three of a kind:  49.0694 %
Straight:  59.493 %
Flush:  62.286 %
Full house:  49.0694 %
Four of a kind:  3.3928 %
Straight flush:  0.1226 %
```

The unique thing I found is that the chances of a flush to exist in a hand is larger than a straight even though the game rule favours the flush to be a higher ranking than the straight. The game rule seems to be following poker rules which only account 7 cards (5 river cards and 2 player cards) that made the chances of straight appearing to be bigger.

V. Conclusion

Combinatorics are incredibly useful for calculating combinations or possibilities in simple scenarios, providing precise results through mathematical equations. However, for more complex problems, such as evaluating all potential outcomes in card games, simulations like the Monte Carlo simulation become essential. These simulations provide approximate but highly accurate results by leveraging random sampling over large datasets. While this discussion focuses specifically on the combinations of cards in a hand, the equations and code provided here are designed to be adaptable and extendable. It is my hope that these tools can serve as a foundation for building an AI model or bot for strategic card games like Big Two, enabling more sophisticated decision-making and gameplay.

VI. Appendix

Source code used for the functions and simulation code of Big Two: https://github.com/BobSwagg13/Application-of-Combinatorics-in-Big-Two

Video link of explaining the code: https://youtu.be/WbO6TasjtX4

VII. Acknowledgment

I would like to express my deepest gratitude to God for the blessings and guidance that enabled me to complete this paper.

My heartfelt thanks go to my family, whose unwavering support and encouragement have been instrumental in making this work possible.

I extend my sincere appreciation to my lecturers, Ir. Rila Mandala, M.Eng., Ph.D., and Dr. Ir. Rinaldi Munir, M.T., for introducing me to the fascinating world of combinatorics and inspiring me throughout this academic journey.

Additionally, I would like to thank my grandfather and friends, who introduced me to the wonderful game that served as the foundation of this study. Their enthusiasm and shared passion for this game were crucial in shaping the ideas presented in this paper.

Finally, I hope that this paper can serve as a valuable reference or tool for others, contributing to the development of similar studies or projects in the future.

REFERENCES

[1] Big2.com.au. (2015). Rules of Big Two. Retrieved December 23, 2024, from https://www.big2.com.au/rules.php . [Accessed: Dec. 24, 2024]..

[2] BobSwagg13, *Application of Combinatorics in Big Two: Analyzing Card Groupings*. GitHub Repository. [Online]. Available: https://github.com/BobSwagg13/Application-of-Combinatorics-in-Big-Two-Analyzing-Card-Groupings- . [Accessed: Dec. 26, 2024].

[3] R. Munir, *Kombinatorika Bagian 1*, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/18-Kombinatorika-Bagian1-2024.pdf. [Accessed: Dec. 24, 2024].

[4] R. Munir, *Kombinatorika Bagian 2,* 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/19-Kombinatorika-Bagian2-2024.pdf. [Accessed: Dec. 24, 2024].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 27 Desember 2024

[illegible]

Bob Kunanda
13523086