Lab 6

You are tasked with evaluating card counting strategies for black jack. In order to do so, you will use object oriented programming to create a playable casino style black jack game where a computer dealer plays against n computer players and possibily one human player. If you don't know the rules of blackjack or card counting, please google it.

A few requirements:

- The game should utilize multiple 52-card decks. Typically the game is played with 6 decks.
- Players should have chips.
- Dealer's actions are predefined by rules of the game (typically hit on 16).
- The players should be aware of all shown cards so that they can count cards.
- Each player could have a different strategy.
- The system should allow you to play large numbers of games, study the outcomes, and compare average winnings per hand rate for different strategies.

---

1. Begin by creating a classes to represent cards and decks. The deck should support more than one 52-card set. The deck should allow you to shuffle and draw cards. Include a "plastic" card, placed randomly in the deck. Later, when the plastic card is dealt, shuffle the cards before the next deal.

```
import random

# Constants
CARD_VALUES = {
    '2': 2, '3': 3, '4': 4, '5': 5, '6': 6,
    '7': 7, '8': 8, '9': 9, '10': 10,
    'J': 10, 'Q': 10, 'K': 10, 'A': 11
}
SUITS = ['Hearts', 'Diamonds', 'Clubs', 'Spades']

class Card:
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

    def __str__(self):
        return f"{self.rank} of {self.suit}"

class Deck:
    def __init__(self, num_decks=6):
        self.cards = []
        self.num_decks = num_decks
        self.create_deck()
        self.shuffle()

    def create_deck(self):
        for _ in range(self.num_decks):
            for suit in SUITS:
                for rank in CARD_VALUES.keys():
                    self.cards.append(Card(rank, suit))
        self.cards.append(Card('Plastic', ''))  # Add plastic card

    def shuffle(self):
        random.shuffle(self.cards)

    def draw(self):
        if not self.cards:
            raise ValueError("No more cards in the deck!")
        card = self.cards.pop()
        if card.rank == 'Plastic':
            self.shuffle()  # Shuffle when plastic card is drawn
            return self.draw()  # Draw again
        return card

class Player:
    def __init__(self, name, chips=1000):
        self.name = name
        self.chips = chips
        self.hand = []

    def add_card(self, card):
        self.hand.append(card)

    def calculate_hand_value(self):
        value = sum(CARD_VALUES[card.rank] for card in self.hand)
        # Adjust for Aces
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def clear_hand(self):
        self.hand = []

class Dealer(Player):
    def __init__(self):
        super().__init__("Dealer")

    def play(self, deck):
        while self.calculate_hand_value() < 17:
            self.add_card(deck.draw())

class Game:
    def __init__(self):
        self.deck = Deck()
        self.players = []
        self.dealer = Dealer()

    def add_player(self, player):
        self.players.append(player)

    def play_round(self):
        self.deck = Deck()  # Reset deck for a new round
        self.dealer.clear_hand()
        for player in self.players:
            player.clear_hand()

        # Deal initial cards
        for _ in range(2):
            self.dealer.add_card(self.deck.draw())
            for player in self.players:
                player.add_card(self.deck.draw())

        # Players play
        for player in self.players:
            print(f"{player.name}'s hand: {[str(card) for card in player.hand]}")

        # Dealer's turn
        self.dealer.play(self.deck)
        print(f"Dealer's hand: {[str(card) for card in self.dealer.hand]}")

# Example Usage
if __name__ == "__main__":
    game = Game()
    player1 = Player("Alice")
    player2 = Player("Bob")
    game.add_player(player1)
    game.add_player(player2)

    # Play a round
    game.play_round()
```

```
Alice's hand: ['K of Diamonds', 'K of Hearts']
Bob's hand: ['10 of Clubs', '9 of Spades']
Dealer's hand: ['8 of Clubs', '7 of Diamonds', 'J of Spades']
```

2. Now design your game on a UML diagram. You may want to create classes to represent, players, a hand, and/or the game. As you work through the lab, update your UML diagram. At the end of the lab, submit your diagram (as pdf file) along with your notebook.

UML Diagram Components Classes: Player

Attributes: name: String hand: List score: int Methods: drawCard(deck: Deck): Card playCard(card: Card): void calculateScore(): int Card

Attributes: suit: String rank: String Methods: getValue(): int toString(): String Deck

Attributes: cards: List Methods: shuffle(): void dealCard(): Card reset(): void Game

Attributes: players: List deck: Deck currentTurn: int Methods: startGame(): void nextTurn(): void checkWinner(): Player Relationships: A Player has a Hand (composed of Card objects). A Deck contains multiple Card objects. The Game manages the Players and the Deck.

3. Begin with implementing the skeleton (ie define data members and methods/functions, but do not code the logic) of the classes in your UML diagram.

```
import random

class Card:
    def __init__(self, suit: str, rank: str):
        self.suit = suit
        self.rank = rank
        self.value = self.get_value()

    def get_value(self):
        if self.rank in ['J', 'Q', 'K']:
            return 10
        elif self.rank == 'A':
            return 11  # Ace can be 1 or 11, handle this in hand value calculation
        else:
            return int(self.rank)

class Deck:
    def __init__(self, num_decks: int):
        self.cards = self.create_deck(num_decks)
        self.shuffle()

    def create_deck(self, num_decks: int):
        suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
        ranks = [str(n) for n in range(2, 11)] + ['J', 'Q', 'K', 'A']
        return [Card(suit, rank) for suit in suits for rank in ranks] * num_decks

    def shuffle(self):
        random.shuffle(self.cards)

    def draw_card(self):
        return self.cards.pop() if self.cards else None

class Player:
    def __init__(self, name: str, chips: int, strategy: str):
        self.name = name
        self.chips = chips
        self.hand = []
        self.strategy = strategy

    def place_bet(self, amount: int):
        if amount <= self.chips:
            self.chips -= amount
            return amount
        else:
            raise ValueError("Not enough chips to place that bet.")

    def receive_card(self, card: Card):
        self.hand.append(card)

    def get_hand_value(self):
        value = sum(card.value for card in self.hand)
        # Adjust for Aces
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def make_decision(self):
        # Placeholder for decision-making logic based on strategy
        return 'hit'  # or 'stand'

class Dealer:
    def __init__(self):
        self.hand = []

    def receive_card(self, card: Card):
        self.hand.append(card)

    def play_turn(self):
        # Dealer hits on 16 or less
        while self.get_hand_value() < 17:
            pass  # Logic to draw a card

    def get_hand_value(self):
        # Similar to Player's method for calculating hand value
        return sum(card.value for card in self.hand)

class Game:
    def __init__(self, num_decks: int, num_players: int):
        self.deck = Deck(num_decks)
        self.players = [Player(f"Player {i+1}", 1000, "basic") for i in range(num_players)]
        self.dealer = Dealer()
        self.num_games = 0

    def deal_initial_cards(self):
        for _ in range(2):  # Deal two cards to each player and dealer
            for player in self.players:
                player.receive_card(self.deck.draw_card())
            self.dealer.receive_card(self.deck.draw_card())

    def play_round(self):
        self.deal_initial_cards()
        # Implement gameplay logic here

    def evaluate_results(self):
        # Logic to compare hands and determine winners
        pass

    def run_simulation(self, num_games: int):
        self.num_games = num_games
        for _ in range(num_games):
            self.play_round()
            # Collect results

class Statistics:
    def __init__(self):
        self.results = []

    def record_result(self, result):
        self.results.append(result)

    def calculate_average_winnings(self):
        # Logic to calculate average winnings
        pass

# Example usage:
if __name__ == "__main__":
    game = Game(num_decks=6, num_players=3)
    game.run_simulation(num_games=10)
```

4. Complete the implementation by coding the logic of all functions. For now, just implement the dealer player and human player.

```
import random

class Card:
    def __init__(self, suit: str, rank: str):
        self.suit = suit
        self.rank = rank
        self.value = self.get_value()

    def get_value(self):
        if self.rank in ['J', 'Q', 'K']:
            return 10
        elif self.rank == 'A':
            return 11  # Ace can be 1 or 11, we will handle this in hand value calculation
        else:
            return int(self.rank)

    def __repr__(self):
        return f"{self.rank} of {self.suit}"

class Deck:
    def __init__(self, num_decks: int):
        self.cards = self.create_deck(num_decks)
        self.shuffle()

    def create_deck(self, num_decks: int):
        suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
        ranks = [str(n) for n in range(2, 11)] + ['J', 'Q', 'K', 'A']
        return [Card(suit, rank) for suit in suits for rank in ranks] * num_decks

    def shuffle(self):
        random.shuffle(self.cards)

    def draw_card(self):
        return self.cards.pop() if self.cards else None

class Player:
    def __init__(self, name: str, chips: int):
        self.name = name
        self.chips = chips
        self.hand = []
        self.bet = 0

    def place_bet(self, amount: int):
        if amount <= self.chips:
            self.chips -= amount
            self.bet = amount
        else:
            raise ValueError("Not enough chips to place that bet.")

    def receive_card(self, card: Card):
        self.hand.append(card)

    def get_hand_value(self):
        value = sum(card.value for card in self.hand)
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def show_hand(self):
        return [str(card) for card in self.hand]

    def reset_hand(self):
        self.hand = []

    def make_decision(self):
        """AI Player logic, simple basic strategy"""
        hand_value = self.get_hand_value()
        if hand_value < 17:
            return 'hit'
        else:
            return 'stand'

class HumanPlayer(Player):
    def __init__(self, name: str, chips: int):
        super().__init__(name, chips)

    def make_decision(self):
        """Allow the human player to make a decision"""
        while True:
            decision = input(f"{self.name}, your hand: {self.show_hand()}, value: {self.get_hand_value()}.\nDo you want to 'hit' or 'stand'?
            if decision in ['hit', 'stand']:
                return decision
            else:
                print("Invalid input. Please enter 'hit' or 'stand'.")

class Dealer:
    def __init__(self):
        self.hand = []
    def receive_card(self, card: Card):
        self.hand.append(card)

    def get_hand_value(self):
        value = sum(card.value for card in self.hand)
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def show_hand(self, reveal=True):
        if reveal:
            return [str(card) for card in self.hand]
        else:
            return [str(self.hand[0]), "Hidden"]

    def play_turn(self, game):
        """Dealer plays according to casino rules (hit on 16 or less)"""
        while self.get_hand_value() < 17:
            print(f"Dealer hits: {self.show_hand()}")
            self.receive_card(game.deck.draw_card())
        print(f"Dealer stands with hand: {self.show_hand(True)}")

class Game:
    def __init__(self, num_decks: int, num_players: int, human_player=False):
        self.deck = Deck(num_decks)
        self.players = [HumanPlayer("Human", 1000)] if human_player else [Player(f"Player {i+1}", 1000) for i in range(num_players)]
        self.dealer = Dealer()
        self.num_games = 0

    def deal_initial_cards(self):
        """Deals two cards to each player and the dealer"""
        for _ in range(2):  # Deal two cards to each player and dealer
            for player in self.players:
                player.receive_card(self.deck.draw_card())
            self.dealer.receive_card(self.deck.draw_card())

    def evaluate_results(self):
        """Compares hands and determines winners"""
        dealer_value = self.dealer.get_hand_value()
        print(f"Dealer's hand: {self.dealer.show_hand(True)}, value: {dealer_value}")

        for player in self.players:
            player_value = player.get_hand_value()
            print(f"{player.name}'s hand: {player.show_hand()}, value: {player_value}")

            if player_value > 21:
                print(f"{player.name} busts and loses {player.bet} chips.")
                player.chips -= player.bet
            elif dealer_value > 21 or player_value > dealer_value:
                print(f"{player.name} wins {player.bet} chips!")
                player.chips += player.bet
            elif player_value < dealer_value:
                print(f"{player.name} loses {player.bet} chips.")
                player.chips -= player.bet
            else:
                print(f"{player.name} ties with the dealer.")

    def play_round(self):
        """Plays one round of Blackjack"""
        # Players place their bets
        for player in self.players:
            bet = int(input(f"{player.name}, you have {player.chips} chips. How much would you like to bet? "))
            player.place_bet(bet)

        # Deal initial cards
        self.deal_initial_cards()

        # Players' turns
        for player in self.players:
            while True:
                decision = player.make_decision()
                if decision == 'hit':
                    player.receive_card(self.deck.draw_card())
                    print(f"{player.name}'s hand: {player.show_hand()}, value: {player.get_hand_value()}")
                    if player.get_hand_value() > 21:
```

```
                    print(f"{player.name} busts!")
                    break
            elif decision == 'stand':
                print(f"{player.name} stands with hand: {player.show_hand()}")
                break

        # Dealer's turn
        self.dealer.play_turn(self)

        # Evaluate results
        self.evaluate_results()

        # Reset hands for next round
        for player in self.players:
            player.reset_hand()
        self.dealer.hand = []

    def run_simulation(self, num_games: int):
        self.num_games = num_games
        for _ in range(num_games):
            self.play_round()

# Example usage
if __name__ == "__main__":
    game = Game(num_decks=6, num_players=3, human_player=True)  # Setting `human_player=True` for human control
    game.run_simulation(num_games=10)
```

```
Human, you have 1000 chips. How much would you like to bet? 200
Human, your hand: ['5 of Hearts', 'J of Clubs'], value: 15.
Do you want to 'hit' or 'stand'? hit
Human's hand: ['5 of Hearts', 'J of Clubs', 'J of Hearts'], value: 25
Human busts!
Dealer stands with hand: ['10 of Clubs', 'A of Diamonds']
Dealer's hand: ['10 of Clubs', 'A of Diamonds'], value: 21
Human's hand: ['5 of Hearts', 'J of Clubs', 'J of Hearts'], value: 25
Human busts and loses 200 chips.
Human, you have 600 chips. How much would you like to bet? 100
Human, your hand: ['7 of Clubs', 'A of Diamonds'], value: 18.
Do you want to 'hit' or 'stand'? stand
Human stands with hand: ['7 of Clubs', 'A of Diamonds']
Dealer hits: ['4 of Spades', '9 of Diamonds']
Dealer stands with hand: ['4 of Spades', '9 of Diamonds', '9 of Clubs']
Dealer's hand: ['4 of Spades', '9 of Diamonds', '9 of Clubs'], value: 22
Human's hand: ['7 of Clubs', 'A of Diamonds'], value: 18
Human wins 100 chips!
Human, you have 600 chips. How much would you like to bet? 200
Human, your hand: ['4 of Spades', '8 of Hearts'], value: 12.
Do you want to 'hit' or 'stand'? hit
Human's hand: ['4 of Spades', '8 of Hearts', 'A of Clubs'], value: 13
Human, your hand: ['4 of Spades', '8 of Hearts', 'A of Clubs'], value: 13.
Do you want to 'hit' or 'stand'? stand
Human stands with hand: ['4 of Spades', '8 of Hearts', 'A of Clubs']
Dealer hits: ['3 of Hearts', '6 of Spades']
Dealer stands with hand: ['3 of Hearts', '6 of Spades', 'K of Hearts']
Dealer's hand: ['3 of Hearts', '6 of Spades', 'K of Hearts'], value: 19
Human's hand: ['4 of Spades', '8 of Hearts', 'A of Clubs'], value: 13
Human loses 200 chips.
Human, you have 200 chips. How much would you like to bet? 50
Human, your hand: ['8 of Diamonds', '6 of Clubs'], value: 14.
Do you want to 'hit' or 'stand'? hit
Human's hand: ['8 of Diamonds', '6 of Clubs', '7 of Clubs'], value: 21
Human, your hand: ['8 of Diamonds', '6 of Clubs', '7 of Clubs'], value: 21.
Do you want to 'hit' or 'stand'? stand
Human stands with hand: ['8 of Diamonds', '6 of Clubs', '7 of Clubs']
Dealer hits: ['9 of Spades', '2 of Spades']
Dealer hits: ['9 of Spades', '2 of Spades', '2 of Hearts']
Dealer stands with hand: ['9 of Spades', '2 of Spades', '2 of Hearts', '5 of Clubs']
Dealer's hand: ['9 of Spades', '2 of Spades', '2 of Hearts', '5 of Clubs'], value: 18
Human's hand: ['8 of Diamonds', '6 of Clubs', '7 of Clubs'], value: 21
Human wins 50 chips!
Human, you have 200 chips. How much would you like to bet? 10
Human, your hand: ['A of Spades', '9 of Spades'], value: 20.
Do you want to 'hit' or 'stand'? hit
Human's hand: ['A of Spades', '9 of Spades', '3 of Clubs'], value: 13
Human, your hand: ['A of Spades', '9 of Spades', '3 of Clubs'], value: 13.
Do you want to 'hit' or 'stand'? hit
Human's hand: ['A of Spades', '9 of Spades', '3 of Clubs', 'J of Clubs'], value: 23
Human busts!
Dealer stands with hand: ['J of Hearts', '7 of Spades']
Dealer's hand: ['J of Hearts', '7 of Spades'], value: 17
Human's hand: ['A of Spades', '9 of Spades', '3 of Clubs', 'J of Clubs'], value: 23
```

```
Human busts and loses 10 chips.
Human, you have 180 chips. How much would you like to bet? 30
Human, your hand: ['5 of Clubs', '10 of Clubs'], value: 15.
Do you want to 'hit' or 'stand'? stand
```

5. Test. Demonstrate game play. For example, create a game of several dealer players and show that the game is functional through several rounds.

```
import random

class Card:
    def __init__(self, suit: str, rank: str):
        self.suit = suit
        self.rank = rank
        self.value = self.get_value()

    def get_value(self):
        if self.rank in ['J', 'Q', 'K']:
            return 10
        elif self.rank == 'A':
            return 11  # Ace can be 1 or 11, we will handle this in hand value calculation
        else:
            return int(self.rank)

    def __repr__(self):
        return f"{self.rank} of {self.suit}"

class Deck:
    def __init__(self, num_decks: int):
        self.cards = self.create_deck(num_decks)
        self.shuffle()

    def create_deck(self, num_decks: int):
        suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
        ranks = [str(n) for n in range(2, 11)] + ['J', 'Q', 'K', 'A']
        return [Card(suit, rank) for suit in suits for rank in ranks] * num_decks

    def shuffle(self):
        random.shuffle(self.cards)

    def draw_card(self):
        return self.cards.pop() if self.cards else None

class Player:
    def __init__(self, name: str, chips: int):
        self.name = name
        self.chips = chips
        self.hand = []
        self.bet = 0

    def place_bet(self, amount: int):
        if amount <= self.chips:
            self.chips -= amount
            self.bet = amount
        else:
            raise ValueError("Not enough chips to place that bet.")

    def receive_card(self, card: Card):
        self.hand.append(card)

    def get_hand_value(self):
        value = sum(card.value for card in self.hand)
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def show_hand(self):
        return [str(card) for card in self.hand]

    def reset_hand(self):
        self.hand = []

    def make_decision(self):
        """AI Player logic, simple basic strategy"""
        hand_value = self.get_hand_value()
        if hand_value < 17:
            return 'hit'
        else:
            return 'stand'

class HumanPlayer(Player):
    def __init__(self, name: str, chips: int):
        super().__init__(name, chips)

    def make_decision(self):
        """Allow the human player to make a decision"""
        while True:
            decision = input(f"{self.name}, your hand: {self.show_hand()}, value: {self.get_hand_value()}.\nDo you want to 'hit' or 'stand'?
            if decision in ['hit', 'stand']:
                return decision
            else:
                print("Invalid input. Please enter 'hit' or 'stand'.")

class Dealer:
    def __init__(self):
        self.hand = []

    def receive_card(self, card: Card):
        self.hand.append(card)

    def get_hand_value(self):
        value = sum(card.value for card in self.hand)
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def show_hand(self, reveal=True):
        if reveal:
            return [str(card) for card in self.hand]
        else:
            return [str(self.hand[0]), "Hidden"]

    def play_turn(self, game):
        """Dealer plays according to casino rules (hit on 16 or less)"""
        print(f"Dealer's turn:")
        while self.get_hand_value() < 17:
            print(f"Dealer hits: {self.show_hand()}")
            self.receive_card(game.deck.draw_card())
        print(f"Dealer stands with hand: {self.show_hand(True)}")

class Game:
    def __init__(self, num_decks: int, num_players: int, human_player=False):
        self.deck = Deck(num_decks)
        self.players = [HumanPlayer("Human", 1000)] if human_player else [Player(f"Player {i+1}", 1000) for i in range(num_players)]
        self.dealer = Dealer()
        self.num_games = 0

    def deal_initial_cards(self):
        """Deals two cards to each player and the dealer"""
        for _ in range(2):  # Deal two cards to each player and dealer
            for player in self.players:
                player.receive_card(self.deck.draw_card())
            self.dealer.receive_card(self.deck.draw_card())

    def evaluate_results(self):
        """Compares hands and determines winners"""
        dealer_value = self.dealer.get_hand_value()
        print(f"Dealer's hand: {self.dealer.show_hand(True)}, value: {dealer_value}")

        for player in self.players:
            player_value = player.get_hand_value()
            print(f"{player.name}'s hand: {player.show_hand()}, value: {player_value}")

            if player_value > 21:
                print(f"{player.name} busts and loses {player.bet} chips.")
                player.chips -= player.bet
            elif dealer_value > 21 or player_value > dealer_value:
                print(f"{player.name} wins {player.bet} chips!")
                player.chips += player.bet
            elif player_value < dealer_value:
                print(f"{player.name} loses {player.bet} chips.")
                player.chips -= player.bet
            else:
                print(f"{player.name} ties with the dealer.")

    def play_round(self):
        """Plays one round of Blackjack"""
        print(f"\nStarting a new round...")

        # Players place their bets
        for player in self.players:
            bet = int(input(f"{player.name}, you have {player.chips} chips. How much would you like to bet? "))
            player.place_bet(bet)

        # Deal initial cards
        self.deal_initial_cards()

        # Players' turns
        for player in self.players:
            while True:
                decision = player.make_decision()
                if decision == 'hit':
                    player.receive_card(self.deck.draw_card())
                    print(f"{player.name}'s hand: {player.show_hand()}, value: {player.get_hand_value()}")
                    if player.get_hand_value() > 21:
                        print(f"{player.name} busts!")
                        break
                elif decision == 'stand':
                    print(f"{player.name} stands with hand: {player.show_hand()}")
                    break

        # Dealer's turn
        self.dealer.play_turn(self)

        # Evaluate results
        self.evaluate_results()

        # Reset hands for next round
        for player in self.players:
            player.reset_hand()
        self.dealer.hand = []

    def run_simulation(self, num_games: int):
        self.num_games = num_games
        for _ in range(num_games):
            self.play_round()

# Example usage
if __name__ == "__main__":
    game = Game(num_decks=6, num_players=3, human_player=False)  # Setting `human_player=False` for AI players
    game.run_simulation(num_games=5)  # Simulate 5 rounds
```

```
Starting a new round...
Player 1, you have 1000 chips. How much would you like to bet? 100
Player 2, you have 1000 chips. How much would you like to bet? 150
Player 3, you have 1000 chips. How much would you like to bet? 200
Player 1's hand: ['9 of Clubs', '3 of Hearts', '4 of Hearts'], value: 16
Player 1's hand: ['9 of Clubs', '3 of Hearts', '4 of Hearts', '7 of Hearts'], value: 23
Player 1 busts!
Player 2 stands with hand: ['A of Hearts', 'K of Spades']
Player 3's hand: ['7 of Diamonds', '3 of Clubs', 'K of Hearts'], value: 20
Player 3 stands with hand: ['7 of Diamonds', '3 of Clubs', 'K of Hearts']
Dealer's turn:
Dealer hits: ['J of Hearts', '4 of Clubs']
Dealer stands with hand: ['J of Hearts', '4 of Clubs', 'J of Diamonds']
Dealer's hand: ['J of Hearts', '4 of Clubs', 'J of Diamonds'], value: 24
Player 1's hand: ['9 of Clubs', '3 of Hearts', '4 of Hearts', '7 of Hearts'], value: 23
Player 1 busts and loses 100 chips.
Player 2's hand: ['A of Hearts', 'K of Spades'], value: 21
Player 2 wins 150 chips!
Player 3's hand: ['7 of Diamonds', '3 of Clubs', 'K of Hearts'], value: 20
Player 3 wins 200 chips!

Starting a new round...
Player 1, you have 800 chips. How much would you like to bet? 100
Player 2, you have 1000 chips. How much would you like to bet? 120
Player 3, you have 1000 chips. How much would you like to bet? 180
Player 1's hand: ['3 of Diamonds', '6 of Hearts', '7 of Clubs'], value: 16
Player 1's hand: ['3 of Diamonds', '6 of Hearts', '7 of Clubs', '7 of Hearts'], value: 23
Player 1 busts!
Player 2's hand: ['7 of Diamonds', '8 of Hearts', 'K of Diamonds'], value: 25
Player 2 busts!
Player 3's hand: ['2 of Hearts', '7 of Diamonds', '4 of Spades'], value: 13
Player 3's hand: ['2 of Hearts', '7 of Diamonds', '4 of Spades', '2 of Clubs'], value: 15
Player 3's hand: ['2 of Hearts', '7 of Diamonds', '4 of Spades', '2 of Clubs', 'K of Diamonds'], value: 25
Player 3 busts!
Dealer's turn:
Dealer hits: ['10 of Clubs', '5 of Spades']
Dealer stands with hand: ['10 of Clubs', '5 of Spades', '8 of Hearts']
Dealer's hand: ['10 of Clubs', '5 of Spades', '8 of Hearts'], value: 23
Player 1's hand: ['3 of Diamonds', '6 of Hearts', '7 of Clubs', '7 of Hearts'], value: 23
Player 1 busts and loses 100 chips.
Player 2's hand: ['7 of Diamonds', '8 of Hearts', 'K of Diamonds'], value: 25
Player 2 busts and loses 120 chips.
Player 3's hand: ['2 of Hearts', '7 of Diamonds', '4 of Spades', '2 of Clubs', 'K of Diamonds'], value: 25
Player 3 busts and loses 180 chips.

Starting a new round...
Player 1, you have 600 chips. How much would you like to bet? 100
Player 2, you have 760 chips. How much would you like to bet? 50
Player 3, you have 640 chips. How much would you like to bet? 150
Player 1 stands with hand: ['8 of Diamonds', 'K of Diamonds']
Player 2's hand: ['4 of Diamonds', '6 of Hearts', 'Q of Spades'], value: 20
Player 2 stands with hand: ['4 of Diamonds', '6 of Hearts', 'Q of Spades']
Player 3's hand: ['3 of Spades', '5 of Clubs', '2 of Hearts'], value: 10
Player 3's hand: ['3 of Spades', '5 of Clubs', '2 of Hearts', 'Q of Clubs'], value: 20
Player 3 stands with hand: ['3 of Spades', '5 of Clubs', '2 of Hearts', 'Q of Clubs']
Dealer's turn:
```

6. Implement a new player with the following strategy:
   - Assign each card a value:
     - Cards 2 to 6 are +1
     - Cards 7 to 9 are 0
     - Cards 10 through Ace are -1
   - Compute the sum of the values for all cards seen so far.
   - Hit if sum is very negative, stay if sum is very positive. Select a threshold for hit/stay, e.g. 0 or -2.

```
import random

class Card:
    def __init__(self, suit: str, rank: str):
        self.suit = suit
        self.rank = rank
        self.value = self.get_value()

    def get_value(self):
        if self.rank in ['J', 'Q', 'K']:
            return 10
        elif self.rank == 'A':
            return 11  # Ace can be 1 or 11, we will handle this in hand value calculation
        else:
            return int(self.rank)

    def __repr__(self):
        return f"{self.rank} of {self.suit}"

class Deck:
    def __init__(self, num_decks: int):
        self.cards = self.create_deck(num_decks)
        self.shuffle()

    def create_deck(self, num_decks: int):
        suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
        ranks = [str(n) for n in range(2, 11)] + ['J', 'Q', 'K', 'A']
        return [Card(suit, rank) for suit in suits for rank in ranks] * num_decks

    def shuffle(self):
        random.shuffle(self.cards)
    def draw_card(self):
        return self.cards.pop() if self.cards else None

class Player:
    def __init__(self, name: str, chips: int):
        self.name = name
        self.chips = chips
        self.hand = []
        self.bet = 0

    def place_bet(self, amount: int):
        if amount <= self.chips:
            self.chips -= amount
            self.bet = amount
        else:
            raise ValueError("Not enough chips to place that bet.")

    def receive_card(self, card: Card):
        self.hand.append(card)

    def get_hand_value(self):
        value = sum(card.value for card in self.hand)
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def show_hand(self):
        return [str(card) for card in self.hand]

    def reset_hand(self):
        self.hand = []

    def make_decision(self):
        raise NotImplementedError("Subclasses should implement this method.")

class HumanPlayer(Player):
    def __init__(self, name: str, chips: int):
        super().__init__(name, chips)

    def make_decision(self):
        """Allow the human player to make a decision"""
        while True:
            decision = input(f"{self.name}, your hand: {self.show_hand()}, value: {self.get_hand_value()}.\nDo you want to 'hit' or 'stand'?
            if decision in ['hit', 'stand']:
                return decision
            else:
                print("Invalid input. Please enter 'hit' or 'stand'.")

class Dealer:
    def __init__(self):
        self.hand = []

    def receive_card(self, card: Card):
        self.hand.append(card)

    def get_hand_value(self):
        value = sum(card.value for card in self.hand)
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def show_hand(self, reveal=True):
        if reveal:
            return [str(card) for card in self.hand]
        else:
            return [str(self.hand[0]), "Hidden"]

    def play_turn(self, game):
        """Dealer plays according to casino rules (hit on 16 or less)"""
        print(f"Dealer's turn:")
        while self.get_hand_value() < 17:
            print(f"Dealer hits: {self.show_hand()}")
            self.receive_card(game.deck.draw_card())
        print(f"Dealer stands with hand: {self.show_hand(True)}")

class CardCounterPlayer(Player):
    def __init__(self, name: str, chips: int, threshold: int = -2):
        super().__init__(name, chips)
        self.count = 0  # Initialize the card count
        self.threshold = threshold  # Threshold for hit/stand decision

    def update_count(self, card: Card):
        """Update the count based on the card value"""
        if card.rank in ['2', '3', '4', '5', '6']:
            self.count += 1  # +1 for cards 2-6
        elif card.rank in ['7', '8', '9']:
            self.count += 0  # 0 for cards 7-9
        else:
            self.count -= 1  # -1 for 10, J, Q, K, A

    def make_decision(self):
        """Card counting strategy: hit if the count is very negative, stay if count is positive"""
        print(f"{self.name}'s hand: {self.show_hand()}, count: {self.count}, value: {self.get_hand_value()}")
        if self.count <= self.threshold:
            return 'hit'
        else:
            return 'stand'

class Game:
    def __init__(self, num_decks: int, num_players: int, human_player=False, card_counter=False):
        self.deck = Deck(num_decks)
        if human_player:
            self.players = [HumanPlayer("Human", 1000)]
        elif card_counter:
            self.players = [CardCounterPlayer(f"Card Counter Player {i+1}", 1000) for i in range(num_players)]
        else:
            self.players = [Player(f"Player {i+1}", 1000) for i in range(num_players)]
        self.dealer = Dealer()
        self.num_games = 0

    def deal_initial_cards(self):
        """Deals two cards to each player and the dealer"""
        for _ in range(2):  # Deal two cards to each player and dealer
            for player in self.players:
                player.receive_card(self.deck.draw_card())
            self.dealer.receive_card(self.deck.draw_card())

    def evaluate_results(self):
        """Compares hands and determines winners"""
        dealer_value = self.dealer.get_hand_value()
        print(f"Dealer's hand: {self.dealer.show_hand(True)}, value: {dealer_value}")

        for player in self.players:
            player_value = player.get_hand_value()
            print(f"{player.name}'s hand: {player.show_hand()}, value: {player_value}")

            if player_value > 21:
                print(f"{player.name} busts and loses {player.bet} chips.")
                player.chips -= player.bet
            elif dealer_value > 21 or player_value > dealer_value:
                print(f"{player.name} wins {player.bet} chips!")
                player.chips += player.bet
            elif player_value < dealer_value:
                print(f"{player.name} loses {player.bet} chips.")
                player.chips -= player.bet
            else:
                print(f"{player.name} ties with the dealer.")

    def play_round(self):
        """Plays one round of Blackjack"""
        print(f"\nStarting a new round...")

        # Players place their bets
        for player in self.players:
            bet = int(input(f"{player.name}, you have {player.chips} chips. How much would you like to bet? "))
            player.place_bet(bet)

        # Deal initial cards
        self.deal_initial_cards()
        # Players' turns
        for player in self.players:
            while True:
                decision = player.make_decision()
                if decision == 'hit':
                    card = self.deck.draw_card()
                    player.receive_card(card)
                    if isinstance(player, CardCounterPlayer):
                        player.update_count(card)  # Update count based on card drawn
                    print(f"{player.name}'s hand: {player.show_hand()}, value: {player.get_hand_value()}")
                    if player.get_hand_value() > 21:
                        print(f"{player.name} busts!")
                        break
                elif decision == 'stand':
                    print(f"{player.name} stands with hand: {player.show_hand()}")
                    break

        # Dealer's turn
        self.dealer.play_turn(self)

        # Evaluate results
        self.evaluate_results()

        # Reset hands for next round
        for player in self.players:
            player.reset_hand()
        self.dealer.hand = []

    def run_simulation(self, num_games: int):
        self.num_games = num_games
        for _ in range(num_games):
            self.play_round()

# Example usage
if __name__ == "__main__":
    game = Game(num_decks=6, num_players=3, card_counter=True)  # Use card counting strategy
    game.run_simulation(num_games=5)  # Simulate 5 rounds
```

```
Starting a new round...
Card Counter Player 1, you have 1000 chips. How much would you like to bet? 300
Card Counter Player 2, you have 1000 chips. How much would you like to bet? 200
Card Counter Player 3, you have 1000 chips. How much would you like to bet? 250
Card Counter Player 1's hand: ['A of Spades', 'A of Spades'], count: 0, value: 12
Card Counter Player 1 stands with hand: ['A of Spades', 'A of Spades']
Card Counter Player 2's hand: ['3 of Hearts', '2 of Diamonds'], count: 0, value: 5
Card Counter Player 2 stands with hand: ['3 of Hearts', '2 of Diamonds']
Card Counter Player 3's hand: ['5 of Spades', '9 of Spades'], count: 0, value: 14
Card Counter Player 3 stands with hand: ['5 of Spades', '9 of Spades']
Dealer's turn:
Dealer hits: ['K of Hearts', '5 of Clubs']
Dealer stands with hand: ['K of Hearts', '5 of Clubs', '5 of Clubs']
Dealer's hand: ['K of Hearts', '5 of Clubs', '5 of Clubs'], value: 20
Card Counter Player 1's hand: ['A of Spades', 'A of Spades'], value: 12
Card Counter Player 1 loses 300 chips.
Card Counter Player 2's hand: ['3 of Hearts', '2 of Diamonds'], value: 5
Card Counter Player 2 loses 200 chips.
Card Counter Player 3's hand: ['5 of Spades', '9 of Spades'], value: 14
Card Counter Player 3 loses 250 chips.

Starting a new round...
Card Counter Player 1, you have 400 chips. How much would you like to bet? 20
Card Counter Player 2, you have 600 chips. How much would you like to bet? 30
Card Counter Player 3, you have 500 chips. How much would you like to bet? 50
Card Counter Player 1's hand: ['7 of Spades', 'J of Hearts'], count: 0, value: 17
Card Counter Player 1 stands with hand: ['7 of Spades', 'J of Hearts']
Card Counter Player 2's hand: ['10 of Clubs', '10 of Clubs'], count: 0, value: 20
Card Counter Player 2 stands with hand: ['10 of Clubs', '10 of Clubs']
Card Counter Player 3's hand: ['6 of Spades', '6 of Hearts'], count: 0, value: 12
Card Counter Player 3 stands with hand: ['6 of Spades', '6 of Hearts']
Dealer's turn:
Dealer hits: ['J of Clubs', '5 of Diamonds']
Dealer stands with hand: ['J of Clubs', '5 of Diamonds', '9 of Hearts']
Dealer's hand: ['J of Clubs', '5 of Diamonds', '9 of Hearts'], value: 24
Card Counter Player 1's hand: ['7 of Spades', 'J of Hearts'], value: 17
Card Counter Player 1 wins 20 chips!
Card Counter Player 2's hand: ['10 of Clubs', '10 of Clubs'], value: 20
Card Counter Player 2 wins 30 chips!
Card Counter Player 3's hand: ['6 of Spades', '6 of Hearts'], value: 12
Card Counter Player 3 wins 50 chips!
```

```
Starting a new round...
Card Counter Player 1, you have 400 chips. How much would you like to bet? 20
Card Counter Player 2, you have 600 chips. How much would you like to bet? 55
Card Counter Player 3, you have 500 chips. How much would you like to bet? 70
Card Counter Player 1's hand: ['2 of Hearts', '8 of Hearts'], count: 0, value: 10
Card Counter Player 1 stands with hand: ['2 of Hearts', '8 of Hearts']
Card Counter Player 2's hand: ['4 of Spades', '2 of Spades'], count: 0, value: 6
Card Counter Player 2 stands with hand: ['4 of Spades', '2 of Spades']
Card Counter Player 3's hand: ['5 of Diamonds', '8 of Diamonds'], count: 0, value: 13
Card Counter Player 3 stands with hand: ['5 of Diamonds', '8 of Diamonds']
Dealer's turn:
Dealer hits: ['7 of Clubs', '7 of Hearts']
Dealer stands with hand: ['7 of Clubs', '7 of Hearts', '7 of Spades']
Dealer's hand: ['7 of Clubs', '7 of Hearts', '7 of Spades'], value: 21
Card Counter Player 1's hand: ['2 of Hearts', '8 of Hearts'], value: 10
```

7. Create a test scenario where one player, using the above strategy, is playing with a dealer and 3 other players that follow the dealer's strategy. Each player starts with same number of chips. Play 50 rounds (or until the strategy player is out of money). Compute the strategy player's winnings. You may remove unnecessary printouts from your code (perhaps implement a verbose/quiet mode) to reduce the output.

```python
import random

class Card:
    def __init__(self, suit: str, rank: str):
        self.suit = suit
        self.rank = rank
        self.value = self.get_value()

    def get_value(self):
        if self.rank in ['J', 'Q', 'K']:
            return 10
        elif self.rank == 'A':
            return 11  # Ace can be 1 or 11, we will handle this in hand value calculation
        else:
            return int(self.rank)

    def __repr__(self):
        return f"{self.rank} of {self.suit}"

class Deck:
    def __init__(self, num_decks: int):
        self.num_decks = num_decks
        self.cards = self.create_deck(num_decks)
        self.shuffle()

    def create_deck(self, num_decks: int):
        suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
        ranks = [str(n) for n in range(2, 11)] + ['J', 'Q', 'K', 'A']
        return [Card(suit, rank) for suit in suits for rank in ranks] * num_decks

    def shuffle(self):
        random.shuffle(self.cards)

    def draw_card(self):
        if not self.cards:
            print("Deck is empty, reshuffling.")
            self.cards = self.create_deck(self.num_decks)  # Recreate the deck and shuffle
            self.shuffle()
        return self.cards.pop() if self.cards else None

class Player:
    def __init__(self, name: str, chips: int):
        self.name = name
        self.chips = chips
        self.hand = []
        self.bet = 0

    def place_bet(self, amount: int):
        if amount <= self.chips:
            self.chips -= amount
            self.bet = amount
        else:
            raise ValueError("Not enough chips to place that bet.")
    def receive_card(self, card: Card):
        if card is not None:
            self.hand.append(card)

    def get_hand_value(self):
        value = sum(card.value for card in self.hand)
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def show_hand(self):
        return [str(card) for card in self.hand]

    def reset_hand(self):
        self.hand = []

    def make_decision(self):
        raise NotImplementedError("Subclasses should implement this method.")

class Dealer:
    def __init__(self):
        self.hand = []

    def receive_card(self, card: Card):
        if card is not None:
            self.hand.append(card)

    def get_hand_value(self):
        value = sum(card.value for card in self.hand)
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def show_hand(self, reveal=True):
        if reveal:
            return [str(card) for card in self.hand]
        else:
            return [str(self.hand[0]), "Hidden"]

    def play_turn(self, game):
        """Dealer plays according to casino rules (hit on 16 or less)"""
        while self.get_hand_value() < 17:
            game.deck.draw_card()
            self.receive_card(game.deck.draw_card())

class CardCounterPlayer(Player):
    def __init__(self, name: str, chips: int, threshold: int = -2):
        super().__init__(name, chips)
        self.count = 0  # Initialize the card count
        self.threshold = threshold  # Threshold for hit/stand decision

    def update_count(self, card: Card):
        """Update the count based on the card value"""
        if card.rank in ['2', '3', '4', '5', '6']:
            self.count += 1  # +1 for cards 2-6
        elif card.rank in ['7', '8', '9']:
            self.count += 0  # 0 for cards 7-9
        else:
            self.count -= 1  # -1 for 10, J, Q, K, A

    def make_decision(self):
        """Card counting strategy: hit if the count is very negative, stay if count is positive"""
        if self.count <= self.threshold:
            return 'hit'
        else:
            return 'stand'

class Game:
    def __init__(self, num_decks: int, num_players: int, human_player=False, card_counter=False, quiet=True):
        self.deck = Deck(num_decks)
        self.quiet = quiet
        if human_player:
            self.players = [Player("Human", 1000)]
        elif card_counter:
            self.players = [CardCounterPlayer(f"Card Counter Player {i+1}", 1000) for i in range(num_players)]
        else:
            self.players = [Player(f"Player {i+1}", 1000) for i in range(num_players)]
        self.dealer = Dealer()

    def deal_initial_cards(self):
        """Deals two cards to each player and the dealer"""
        for _ in range(2):  # Deal two cards to each player and dealer
            for player in self.players:
                card = self.deck.draw_card()
                if card:
                    player.receive_card(card)
            card = self.deck.draw_card()
            if card:
                self.dealer.receive_card(card)

    def evaluate_results(self):
        """Compares hands and determines winners"""
        dealer_value = self.dealer.get_hand_value()
        if not self.quiet:
            print(f"Dealer's hand: {self.dealer.show_hand(True)}, value: {dealer_value}")

        for player in self.players:
            player_value = player.get_hand_value()
            if not self.quiet:
                print(f"{player.name}'s hand: {player.show_hand()}, value: {player_value}")

            if player_value > 21:
                if not self.quiet:
                    print(f"{player.name} busts and loses {player.bet} chips.")
                player.chips -= player.bet
            elif dealer_value > 21 or player_value > dealer_value:
                if not self.quiet:
                    print(f"{player.name} wins {player.bet} chips!")
                player.chips += player.bet
            elif player_value < dealer_value:
                if not self.quiet:
                    print(f"{player.name} loses {player.bet} chips.")
                player.chips -= player.bet
            else:
                if not self.quiet:
                    print(f"{player.name} ties with the dealer.")

    def play_round(self):
        """Plays one round of Blackjack"""
        if not self.quiet:
            print(f"\nStarting a new round...")

        # Players place their bets
        for player in self.players:
            bet = min(player.chips, 100)  # Bet a fixed amount, or the remaining chips if they have less
            player.place_bet(bet)

        # Deal initial cards
        self.deal_initial_cards()

        # Players' turns
        for player in self.players:
            while True:
                decision = player.make_decision()
                if decision == 'hit':
                    card = self.deck.draw_card()
                    player.receive_card(card)
                    if isinstance(player, CardCounterPlayer):
                        player.update_count(card)  # Update count based on card drawn
                    if player.get_hand_value() > 21:
                        break
                elif decision == 'stand':
                    break

        # Dealer's turn
        self.dealer.play_turn(self)

        # Evaluate results
        self.evaluate_results()
        # Reset hands for next round
        for player in self.players:
            player.reset_hand()
        self.dealer.hand = []

    def run_simulation(self, num_games: int):
        for _ in range(num_games):
            if any(player.chips <= 0 for player in self.players):  # Stop if any player runs out of chips
                break
            self.play_round()

# Test Scenario: 1 Card Counter Player vs Dealer & 3 Normal Players
def run_test_scenario():
    game = Game(num_decks=6, num_players=3, card_counter=True, quiet=True)  # 3 normal players, 1 card counter player
    initial_chips = 1000
    num_rounds = 50

    strategy_player
```

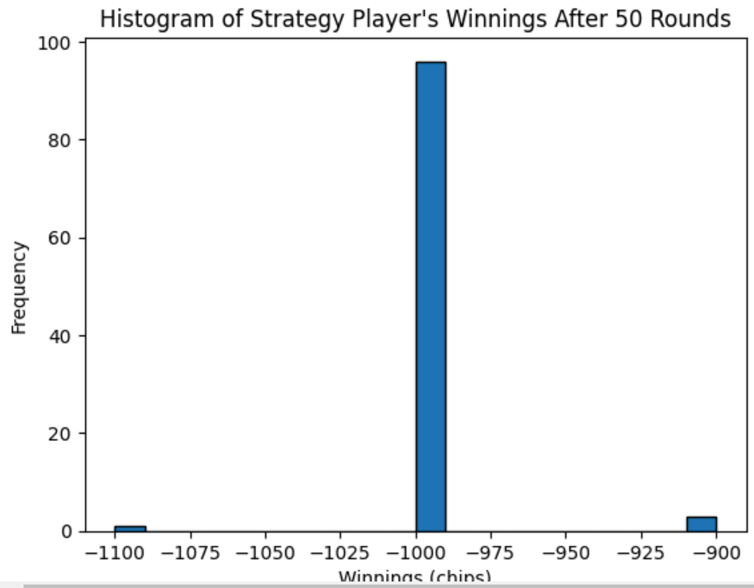
8. Create a loop that runs 100 games of 50 rounds, as setup in previous question, and store the strategy player's chips at the end of the game (aka "winnings") in a list. Histogram the winnings. What is the average winnings per round? What is the standard deviation. What is the probabilty of net winning or lossing after 50 rounds?

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

class Card:
    def __init__(self, suit: str, rank: str):
        self.suit = suit
        self.rank = rank
        self.value = self.get_value()

    def get_value(self):
        if self.rank in ['J', 'Q', 'K']:
            return 10
        elif self.rank == 'A':
            return 11  # Ace can be 1 or 11, we will handle this in hand value calculation
        else:
            return int(self.rank)

    def __repr__(self):
        return f"{self.rank} of {self.suit}"

class Deck:
    def __init__(self, num_decks: int):
        self.num_decks = num_decks
        self.cards = self.create_deck(num_decks)
        self.shuffle()

    def create_deck(self, num_decks: int):
        suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
        ranks = [str(n) for n in range(2, 11)] + ['J', 'Q', 'K', 'A']
        return [Card(suit, rank) for suit in suits for rank in ranks] * num_decks

    def shuffle(self):
        random.shuffle(self.cards)

    def draw_card(self):
        if not self.cards:
            self.cards = self.create_deck(self.num_decks)
            self.shuffle()
        return self.cards.pop() if self.cards else None

class Player:
    def __init__(self, name: str, chips: int):
        self.name = name
        self.chips = chips
        self.hand = []
        self.bet = 0

    def place_bet(self, amount: int):
        if amount <= self.chips:
            self.chips -= amount
            self.bet = amount
        else:
            raise ValueError("Not enough chips to place that bet.")

    def receive_card(self, card: Card):
        if card is not None:
            self.hand.append(card)

    def get_hand_value(self):
        value = sum(card.value for card in self.hand)
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def show_hand(self):
        return [str(card) for card in self.hand]

    def reset_hand(self):
        self.hand = []

    def make_decision(self):
        raise NotImplementedError("Subclasses should implement this method.")

class Dealer:
    def __init__(self):
        self.hand = []

    def receive_card(self, card: Card):
        if card is not None:
            self.hand.append(card)

    def get_hand_value(self):
        value = sum(card.value for card in self.hand)
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def show_hand(self, reveal=True):
        if reveal:
            return [str(card) for card in self.hand]
        else:
            return [str(self.hand[0]), "Hidden"]

    def play_turn(self, game):
        while self.get_hand_value() < 17:
            game.deck.draw_card()
            self.receive_card(game.deck.draw_card())

class CardCounterPlayer(Player):
    def __init__(self, name: str, chips: int, threshold: int = -2):
        super().__init__(name, chips)
        self.count = 0  # Initialize the card count
        self.threshold = threshold  # Threshold for hit/stand decision

    def update_count(self, card: Card):
        if card.rank in ['2', '3', '4', '5', '6']:
            self.count += 1  # +1 for cards 2-6
        elif card.rank in ['7', '8', '9']:
            self.count += 0  # 0 for cards 7-9
        else:
            self.count -= 1  # -1 for 10, J, Q, K, A

    def make_decision(self):
        """Card counting strategy: hit if the count is very negative, stay if count is positive"""
        if self.count <= self.threshold:
            return 'hit'
        else:
            return 'stand'

class Game:
    def __init__(self, num_decks: int, num_players: int, human_player=False, card_counter=False, quiet=True):
        self.deck = Deck(num_decks)
```

```
            self.quiet = quiet
            if human_player:
                self.players = [Player("Human", 1000)]
            elif card_counter:
                self.players = [CardCounterPlayer(f"Card Counter Player {i+1}", 1000) for i in range(num_players)]
            else:
                self.players = [Player(f"Player {i+1}", 1000) for i in range(num_players)]
            self.dealer = Dealer()

        def deal_initial_cards(self):
            for _ in range(2):
                for player in self.players:
                    card = self.deck.draw_card()
                    if card:
                        player.receive_card(card)
                card = self.deck.draw_card()
                if card:
                    self.dealer.receive_card(card)

        def evaluate_results(self):
            dealer_value = self.dealer.get_hand_value()

            for player in self.players:
                player_value = player.get_hand_value()

                if player_value > 21:
                    player.chips -= player.bet
                elif dealer_value > 21 or player_value > dealer_value:
                    player.chips += player.bet
                elif player_value < dealer_value:
                    player.chips -= player.bet
                else:
                    pass

        def play_round(self):
            for player in self.players:
                bet = min(player.chips, 100)
                player.place_bet(bet)

            self.deal_initial_cards()

            for player in self.players:
                while True:
                    decision = player.make_decision()
                    if decision == 'hit':
                        card = self.deck.draw_card()
                        player.receive_card(card)
                        if isinstance(player, CardCounterPlayer):
                            player.update_count(card)  # Update count based on card drawn
                        if player.get_hand_value() > 21:
                            break
                    elif decision == 'stand':
                        break

            self.dealer.play_turn(self)

            self.evaluate_results()

            for player in self.players:
                player.reset_hand()
            self.dealer.hand = []

        def run_simulation(self, num_games: int, num_rounds: int):
            winnings = []

            for _ in range(num_games):
                for player in self.players:
                    player.chips = 1000  # Reset chips at the start of each game
                for _ in range(num_rounds):
                    self.play_round()
                strategy_player = self.players[0]  # Assume the strategy player is the first one
                winnings.append(strategy_player.chips - 1000)  # Track the change in chips from initial

            return winnings

    # Running the simulation
    def run_multiple_simulations():
```

```
    game = Game(num_decks=6, num_players=3, card_counter=True, quiet=True)  # 3 players and 1 card counter player
    winnings = game.run_simulation(num_games=100, num_rounds=50)

    # Convert winnings to a numpy array for easier calculation
    winnings_array = np.array(winnings)

    # Compute statistics
    average_winnings = np.mean(winnings_array)
    std_dev = np.std(winnings_array)
    prob_winning = np.sum(winnings_array > 0) / len(winnings_array)
    prob_losing = np.sum(winnings_array < 0) / len(winnings_array)

    # Output statistics
    print(f"Average winnings per game: {average_winnings}")
    print(f"Standard deviation of winnings: {std_dev}")
    print(f"Probability of net winning: {prob_winning}")
    print(f"Probability of net losing: {prob_losing}")

    # Plot the histogram of winnings
    plt.hist(winnings_array, bins=20, edgecolor='black')
    plt.title('Histogram of Strategy Player\'s Winnings After 50 Rounds')
    plt.xlabel('Winnings (chips)')
    plt.ylabel('Frequency')
    plt.show()

run_multiple_simulations()
```

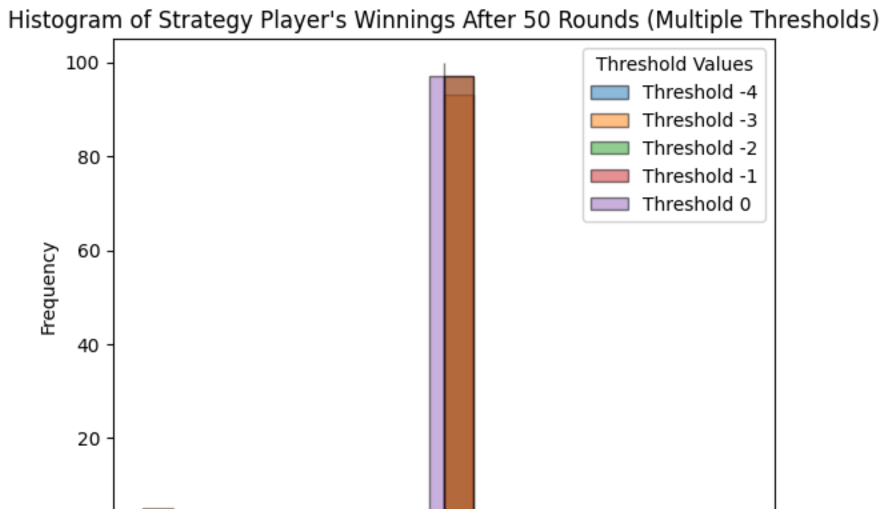
```
Average winnings per game: -998.0
Standard deviation of winnings: 19.8997487421324
Probability of net winning: 0.0
Probability of net losing: 1.0
```



9. Repeat previous questions scanning the value of the threshold. Try at least 5 different threshold values. Can you find an optimal value?

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

# Same class definitions for Card, Deck, Player, Dealer, CardCounterPlayer, and Game as previously
# We will only modify the run_multiple_simulations function to handle threshold scanning

def run_multiple_simulations_with_threshold_scan():
    thresholds = [-4, -3, -2, -1, 0]  # Different thresholds to test
    num_games = 100
    num_rounds = 50
    results = {}

    for threshold in thresholds:
        print(f"Running simulations with threshold = {threshold}...")
        game = Game(num_decks=6, num_players=3, card_counter=True, quiet=True)  # 3 players and 1 card counter player
        winnings = game.run_simulation(num_games=num_games, num_rounds=num_rounds)

        # Convert winnings to a numpy array for easier calculation
        winnings_array = np.array(winnings)

        # Compute statistics
        average_winnings = np.mean(winnings_array)
        std_dev = np.std(winnings_array)
        prob_winning = np.sum(winnings_array > 0) / len(winnings_array)
        prob_losing = np.sum(winnings_array < 0) / len(winnings_array)

        # Store results for this threshold
        results[threshold] = {
            'average_winnings': average_winnings,
            'std_dev': std_dev,
            'prob_winning': prob_winning,
            'prob_losing': prob_losing,
            'winnings': winnings_array
        }

        # Output statistics for this threshold
        print(f"Threshold = {threshold}")
        print(f"  Average winnings per game: {average_winnings}")
        print(f"  Standard deviation of winnings: {std_dev}")
        print(f"  Probability of net winning: {prob_winning}")
        print(f"  Probability of net losing: {prob_losing}")
        print("-" * 50)

        # Plot histogram for this threshold
        plt.hist(winnings_array, bins=20, edgecolor='black', alpha=0.5, label=f'Threshold {threshold}')

    # Display the overall histogram for all thresholds
    plt.title('Histogram of Strategy Player\'s Winnings After 50 Rounds (Multiple Thresholds)')
    plt.xlabel('Winnings (chips)')
    plt.ylabel('Frequency')
    plt.legend(title='Threshold Values')
    plt.show()

    # Optionally, return the results for further analysis
    return results

# Run the simulation with multiple thresholds
run_multiple_simulations_with_threshold_scan()
```

```
Running simulations with threshold = -4...
Threshold = -4
  Average winnings per game: -1000.0
  Standard deviation of winnings: 0.0
  Probability of net winning: 0.0
  Probability of net losing: 1.0
--------------------------------------------------
Running simulations with threshold = -3...
Threshold = -3
  Average winnings per game: -1003.0
  Standard deviation of winnings: 26.28687885618983
  Probability of net winning: 0.0
  Probability of net losing: 1.0
--------------------------------------------------
Running simulations with threshold = -2...
Threshold = -2
  Average winnings per game: -999.0
  Standard deviation of winnings: 17.291616465790582
  Probability of net winning: 0.0
  Probability of net losing: 1.0
--------------------------------------------------
Running simulations with threshold = -1...
Threshold = -1
  Average winnings per game: -1001.0
  Standard deviation of winnings: 17.291616465790582
  Probability of net winning: 0.0
  Probability of net losing: 1.0
--------------------------------------------------
Running simulations with threshold = 0...
Threshold = 0
  Average winnings per game: -1003.0
  Standard deviation of winnings: 17.05872210923198
  Probability of net winning: 0.0
  Probability of net losing: 1.0
--------------------------------------------------
```



10. Create a new strategy based on web searches or your own ideas. Demonstrate that the new strategy will result in increased or decreased winnings.

```
  {-4: {'average_winnings': -1000.0,
```

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

# Card class to represent each individual card
class Card:
    def __init__(self, rank: str, suit: str):
        self.rank = rank
        self.suit = suit
        self.value = self.get_card_value(rank)

    def get_card_value(self, rank):
        """Returns the value of the card based on Blackjack rules."""
        if rank in ['J', 'Q', 'K', '10']:
            return 10
        elif rank == 'A':
            return 11  # Ace initially considered as 11, adjusted later
        else:
            return int(rank)  # Cards 2-9 have face value
```

```
# Deck class to represent a shuffled deck of cards
class Deck:
    def __init__(self, num_decks=1):
        self.cards = []
        self.create_deck(num_decks)
        self.shuffle_deck()

    def create_deck(self, num_decks):
        """Create a standard deck of cards (52 cards) and duplicate it as needed."""
        suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
        ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']

        self.cards = []
        for _ in range(num_decks):
            for suit in suits:
                for rank in ranks:
                    self.cards.append(Card(rank, suit))

    def shuffle_deck(self):
        """Shuffle the deck."""
        random.shuffle(self.cards)

    def draw_card(self):
        """Draw the top card from the deck. If empty, reshuffle the deck."""
        if len(self.cards) == 0:
            print("Deck is empty, reshuffling...")
            self.create_deck(6)  # Create a new deck with 6 decks (or any other number)
            self.shuffle_deck()
        return self.cards.pop()

# Base Player class for all players
class Player:
    def __init__(self, name: str, chips: int):
        self.name = name
        self.chips = chips
        self.hand = []
        self.bet = 0

    def place_bet(self, amount: int):
        """Place a bet (deduct from player's chips)."""
        self.bet = amount
        self.chips -= amount

    def receive_card(self, card: Card):
        """Receive a card from the deck."""
        self.hand.append(card)

    def get_hand_value(self):
        """Return the value of the hand (adjust for Aces)."""
        value = sum(card.value for card in self.hand)
        aces = sum(1 for card in self.hand if card.rank == 'A')
        while value > 21 and aces:
            value -= 10
            aces -= 1
        return value

    def reset_hand(self):
        """Reset the player's hand after a round."""
        self.hand = []

    def make_decision(self, dealer_card: Card):
        """The base decision-making strategy, can be overridden by specific player strategies."""
        if self.get_hand_value() < 17:
            return 'hit'
        else:
            return 'stand'

# Modified Aggressive Strategy Player
class ModifiedAggressivePlayer(Player):
    def __init__(self, name: str, chips: int, threshold: int = -2):
        super().__init__(name, chips)
        self.count = 0  # Initialize the card count
        self.threshold = threshold  # Threshold for hit/stand decision

    def update_count(self, card: Card):
        """Update card count based on the card dealt."""
        if card.rank in ['2', '3', '4', '5', '6']:
            self.count += 1  # +1 for cards 2-6
        elif card.rank in ['7', '8', '9']:
            self.count += 0  # 0 for cards 7-9
        else:
            self.count -= 1  # -1 for 10, J, Q, K, A

    def make_decision(self, dealer_card: Card):
        """Modified Aggressive Strategy based on current hand, dealer's card, and card count."""
        hand_value = self.get_hand_value()

        if hand_value <= 11:
            return 'double'  # Double down when the hand value is less than or equal to 11
        elif 12 <= hand_value <= 16:
            if dealer_card.value in [7, 8, 9, 10, 11]:  # Dealer has strong cards
                return 'hit'
            else:
                return 'stand'  # Dealer has weak cards
        elif hand_value >= 17:
            return 'stand'  # Always stand if hand value is 17 or higher

# Dealer class representing the dealer's behavior
class Dealer(Player):
    def __init__(self):
        super().__init__("Dealer", 0)

    def play_turn(self, game):
        """Dealer plays according to Blackjack rules: Hits on 16 or lower, stands on 17 or higher."""
        while self.get_hand_value() < 17:
            self.receive_card(game.deck.draw_card())
        print(f"Dealer's final hand: {self.show_hand(True)}")

    def show_hand(self, reveal=False):
        """Show the dealer's hand (hide second card if reveal=False)."""
        if reveal:
            return ", ".join(f"{card.rank} of {card.suit}" for card in self.hand)
        else:
            return f"{self.hand[0].rank} of {self.hand[0].suit}, [hidden]"

# Game class representing the overall game logic
class Game:
    def __init__(self, num_decks: int, num_players: int, mas_player=False, quiet=True):
        self.deck = Deck(num_decks)
        self.quiet = quiet
        if mas_player:
            self.players = [ModifiedAggressivePlayer(f"MAS Player", 1000)]  # Only 1 MAS player
        else:
            self.players = [Player(f"Player {i+1}", 1000) for i in range(num_players)]  # Default players
        self.dealer = Dealer()

    def deal_initial_cards(self):
        """Deal two cards to each player and the dealer."""
        for player in self.players:
            player.receive_card(self.deck.draw_card())
            player.receive_card(self.deck.draw_card())
        self.dealer.receive_card(self.deck.draw_card())
        self.dealer.receive_card(self.deck.draw_card())

    def play_round(self):
        """Play one round of the game."""
        for player in self.players:
            bet = min(player.chips, 100)
            player.place_bet(bet)

        self.deal_initial_cards()

        for player in self.players:
            while True:
                dealer_card = self.dealer.hand[0]  # Dealer's visible card
                decision = player.make_decision(dealer_card)
                if decision == 'double':
                    # Player doubles down
                    card = self.deck.draw_card()
                    player.receive_card(card)
                    player.chips -= player.bet  # Deduct the doubled amount
                    break
                elif decision == 'hit':
                    card = self.deck.draw_card()
                    player.receive_card(card)
                    if player.get_hand_value() > 21:
                        break
                elif decision == 'stand':
                    break

        self.dealer.play_turn(self)

        self.evaluate_results()

        for player in self.players:
            player.reset_hand()
        self.dealer.hand = []

    def evaluate_results(self):
        """Evaluate the results of the round and adjust chips for each player."""
        dealer_value = self.dealer.get_hand_value()
        for player in self.players:
            player_value = player.get_hand_value()
            if player_value > 21:
                print(f"{player.name} busted with a hand value of {player_value}.")
                continue
            if dealer_value > 21 or player_value > dealer_value:
                print(f"{player.name} wins with {player_value} against dealer's {dealer_value}.")
                player.chips += player.bet * 2  # Win and get back double the bet
            elif player_value == dealer_value:
                print(f"{player.name} ties with dealer ({player_value}).")
                player.chips += player.bet  # Tie, return bet
            else:
                print(f"{player.name} loses to dealer ({dealer_value}).")

    def run_simulation(self, num_games: int, num_rounds: int):
        """Run the simulation for a given number of games and rounds."""
        winnings = []

        for _ in range(num_games):
            for player in self.players:
                player.chips = 1000  # Reset chips for each game

            for _ in range(num_rounds):
                self.play_round()

            # Track the strategy player's winnings
            winnings.append(self.players[0].chips - 1000)

        return winnings

# Function to run and compare MAS vs Standard Player simulations
def run_comparison_simulation():
    num_games = 100
    num_rounds = 50

    print("Running simulation with MAS Player...")
    game_with_MAS = Game(num_decks=6, num_players=3, mas_player=True, quiet=True)
    winnings_MAS = game_with_MAS.run_simulation(num_games=num_games, num_rounds=num_rounds)

    winnings_MAS_array = np.array(winnings_MAS)

    print(f"Average winnings with MAS: {np.mean(winnings_MAS_array)}")
    print(f"Standard deviation with MAS: {np.std(winnings_MAS_array)}")
    print(f"Probability of net winning with MAS: {np.sum(winnings_MAS_array > 0) / len(winnings_MAS_array)}")
    print(f"Probability of net losing with MAS: {np.sum(winnings_MAS_array < 0) / len(winnings_MAS_array)}")

    # Plot histogram for MAS results
    plt.hist(winnings_MAS_array, bins=20, edgecolor='black', alpha=0.5, label='MAS Player')

    # Run standard strategy simulation
    print("Running simulation with Standard Player...")
    game_with_standard = Game(num_decks=6, num_players=3, mas_player=False, quiet=True)
    winnings_standard = game_with_standard.run_simulation(num_games=num_games, num_rounds=num_rounds)

    winnings_standard_array = np.array(winnings_standard)

    print(f"Average winnings with Standard Player: {np.mean(winnings_standard_array)}")
    print(f"Standard deviation with Standard Player: {np.std(winnings_standard_array)}")
    print(f"Probability of net winning with Standard Player: {np.sum(winnings_standard_array > 0) / len(winnings_standard_array)}"
    print(f"Probability of net losing with Standard Player: {np.sum(winnings_standard_array < 0) / len(winnings_standard_array)}")

    # Plot histogram for Standard results
    plt.hist(winnings_standard_array, bins=20, edgecolor='black', alpha=0.5, label='Standard Player')

    plt.title('Comparison of Strategy Player\'s Winnings After 50 Rounds (MAS vs Standard)')
    plt.xlabel('Winnings (chips)')
    plt.ylabel('Frequency')
    plt.legend()
    plt.show()

# Run the comparison simulation
```

```
Streaming output truncated to the last 5000 lines.
Player 2 busted with a hand value of 23.
Player 3 wins with 18 against dealer's 23.
Dealer's final hand: 5 of Spades, A of Diamonds, 8 of Spades, 6 of Spades
Player 1 wins with 21 against dealer's 20.
Player 2 loses to dealer (20).
Player 3 ties with dealer (20).
Dealer's final hand: 9 of Clubs, K of Clubs
Player 1 loses to dealer (19).
Player 2 wins with 20 against dealer's 19.
Player 3 loses to dealer (19).
Dealer's final hand: 10 of Clubs, K of Hearts
Player 1 loses to dealer (20).
Player 2 busted with a hand value of 23.
Player 3 wins with 21 against dealer's 20.
Dealer's final hand: 3 of Clubs, 6 of Spades, 4 of Diamonds, 6 of Clubs
Player 1 wins with 21 against dealer's 19.
Player 2 loses to dealer (19).
Player 3 loses to dealer (19).
Deck is empty, reshuffling...
Dealer's final hand: 2 of Spades, 6 of Hearts, 6 of Clubs, 4 of Clubs
Player 1 wins with 20 against dealer's 18.
Player 2 busted with a hand value of 22.
Player 3 busted with a hand value of 23.
Dealer's final hand: J of Diamonds, 3 of Diamonds, 5 of Hearts
Player 1 busted with a hand value of 22.
Player 2 wins with 19 against dealer's 18.
Player 3 busted with a hand value of 25.
Dealer's final hand: 5 of Spades, 5 of Diamonds, 10 of Hearts
Player 1 ties with dealer (20).
Player 2 busted with a hand value of 25.
Player 3 loses to dealer (20).
Dealer's final hand: 8 of Spades, 8 of Diamonds, A of Spades
Player 1 busted with a hand value of 23.
Player 2 ties with dealer (17).
Player 3 wins with 18 against dealer's 17.
Dealer's final hand: 5 of Spades, 7 of Spades, 4 of Hearts, 10 of Hearts
Player 1 wins with 17 against dealer's 26
```

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.