Zeru-Zhou-project12(1)

April 15, 2022

1 Project 12 – Zeru Zhou

TA Help: NA

Collaboration: NA

- Get help from dr. Ward's videos

1.1 Question 1

[1]:
```
import random
from collections import Counter
import pandas as pd
import numpy as np

class Player:

    def __init__(self, name, strategy):
        self.name = name
        self.hand = []
        self.strategy = strategy

    def __str__(self):
        return self.name

    def draw(self):
        self.strategy.draw()

    def discard(self):
        self.strategy.discard()

    def can_end_game(self):
        return self.strategy.can_end_game(self)

    def should_end_game(self):
        return self.strategy.should_end_game(self)

    def make_move(self, game):
```

```
        return self.strategy.make_move(self, game)

    def get_best_hand(self):
        return self.strategy.get_best_hand(self)
```

This code is closer to inheritance. This is because it is derived from the "Strategy" class and could be counted as a subclass of "strategy", but not "has a" component.

1.2 Question 2

1: Game class and Player class because in init function of Game class, it assign self.player to players originated from Player class. 2: Game class and Ruleset class because in init function of Game class, it defined self.ruleset as some provided ruleset by Ruleset class. 3: Game class and Scorecard class because in init function of Game class, it assigned self.scorecard to provided scorecard from Scorecard class.

1.3 Question 3

[2]:
```
import random
from collections import Counter
import pandas as pd
import numpy as np


class Card:
    _value_dict = {"2": 2, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7, "8":8, "9":
↪9, "10": 10, "j": 11, "q": 12, "k": 13, "a": 1}
    _gin_value_dict = {"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": 1}
    def __init__(self, number, suit):
        if str(number).lower() not in [str(num) for num in range(2, 11)] +␣
↪list("jqka"):
            raise Exception("Number wasn't 2-10 or J, Q, K, or A.")
        else:
            self.number = str(number).lower()
        if suit.lower() not in ["clubs", "hearts", "diamonds", "spades"]:
            raise Exception("Suit wasn't one of: clubs, hearts, spades, or␣
↪diamonds.")
        else:
            self.suit = suit.lower()

    def __str__(self):
        return(f'{self.number} of {self.suit.lower()}')

    def __repr__(self):
        return(f'Card(str({self.number}), "{self.suit}")')
```

```
    def __eq__(self, other):
        if self.number == other.number:
            return True
        else:
            return False

    def __lt__(self, other):
        if self._value_dict[self.number] < self._value_dict[other.number]:
            return True
        else:
            return False

    def __gt__(self, other):
        if self._value_dict[self.number] > self._value_dict[other.number]:
            return True
        else:
            return False

    def __hash__(self):
        return hash(self.number)


class Deck:
    brand = "Bicycle"
    _suits = ["clubs", "hearts", "diamonds", "spades"]
    _numbers = [str(num) for num in range(2, 11)] + list("jqka")

    def __init__(self):
        self.cards = [Card(number, suit) for suit in self._suits for number in␣
 ↪self._numbers]

    def __len__(self):
        return len(self.cards)

    def __getitem__(self, key):
        return self.cards[key]

    def __setitem__(self, key, value):
        self.cards[key] = value

    def __str__(self):
        return f"A {self.brand.lower()} deck."


class Player:

    def __init__(self, name, strategy):
```

```
        self.name = name
        self.hand = []
        self.strategy = strategy

    def __str__(self):
        return self.name

    def draw(self):
        self.strategy.draw()

    def discard(self):
        self.strategy.discard()

    def can_end_game(self):
        return self.strategy.can_end_game(self)

    def should_end_game(self):
        return self.strategy.should_end_game(self)

    def make_move(self, game):
        return self.strategy.make_move(self, game)

    def get_best_hand(self):
        return self.strategy.get_best_hand(self)

    def hand_as_df(self, my_cards=None):
        if not my_cards:
            my_cards = self.hand

        data = {'suit': [], 'numeric_value': [], 'card': []}
        for card in my_cards:
            data['suit'].append(card.suit)
            data['numeric_value'].append(card._value_dict[card.number])
            data['card'].append(card)

        return pd.DataFrame(data=data)

    def get_sets(self, my_cards=None):

        if not my_cards:
            my_cards = self.hand

        def _flatten(t):
            return [item for sublist in t for item in sublist]

        def _get_cards_with_value(card_with_value, my_cards):
            return [card for card in my_cards if card == card_with_value]
```

```
        summarized = Counter(my_cards)
        sets = []
        for key, value in summarized.items():
            if value > 2:
                sets.append(_get_cards_with_value(key, my_cards))

        set_tuples = [(x._value_dict[x.number], x.suit) for x in _flatten(sets)]
        remaining_cards = list(filter(lambda x: (x._value_dict[x.number], x.
↪suit) not in set_tuples, my_cards))

        return remaining_cards, sets

    def get_runs(self, my_cards=None):

        if not my_cards:
            my_cards = self.hand

        def _flatten(t):
            return [item for sublist in t for item in sublist]

        # get the hand as a pandas df
        df = self.hand_as_df(my_cards)

        # to store complete runs
        runs = []

        # loop through cards by suit
        for _, group in df.groupby("suit"):

            # sort the sub dataframe, group, by numeric value
            sorted_values = group.sort_values(["numeric_value"])

            # this is the key. create an auxilliary column that
            # is the difference between a column containing a count,
            # for example, 1, 2, 3, 4, 5, and the corresponding
            # numeric_values. This gives us a value that we can group by
            # containing all of the values in a run!
            sorted_values['aux'] = np.
↪arange(len(sorted_values['numeric_value'])) - sorted_values['numeric_value']

            # sub groups here, subdf, will only contain runs now
            for _, subdf in sorted_values.groupby('aux'):

                # if the run is more than 2
                if subdf.shape[0] > 2:
```

```
                    # add the card objects to our list of lists
                    runs.append(subdf['card'].tolist())

        run_tuples = [(x._value_dict[x.number], x.suit) for x in _flatten(runs)]

        remaining_cards = list(filter(lambda x: (x._value_dict[x.number], x.
↪suit) not in run_tuples, my_cards))

        return remaining_cards, runs


class Ruleset:

    @staticmethod
    def deal(game):
        """
        This implementation of deal we will deal
        10 cards each, alternating, starting
        with player1.

        Note: We are _not_ using our strategy to
        draw cards, but rather just drawing 10 cards
        each from the game's deck.
        """
        for _ in range(10):
            card = game.deck.cards.pop(0)
            game.player1.hand.append(card)

            card = game.deck.cards.pop(0)
            game.player2.hand.append(card)

    @staticmethod
    def first_move(game):
        """
        This implementation of first move
        will randomly choose a player to start,
        that player will draw, discard, etc.

        Afterwords, it will return two values. The
        first is a boolean indicating whether or not
        to end the game. The second is the player object.

        If the boolean indicates to end the game the player
        is the player ending the game, otherwise, it is
        the player whose turn is next.
        """
        player_to_start = random.choice((game.player1, game.player2))
```

```
        return player_to_start.make_move(game)


class Strategy:

    @staticmethod
    def get_best_hand(player):

        def _flatten(t):
            return [item for sublist in t for item in sublist]

        # this strategy is to get the runs then sets in that order,
        # count the remaining card values, then reverse the process,
        # get the sets then runs in that order, then count remaining
        # card values
        remaining_1 = player.hand
        remaining_1, runs1 = player.get_runs()
        remaining_1, sets1 = player.get_sets(remaining_1)

        remaining_card_value_1 = 0
        for card in remaining_1:
            remaining_card_value_1 += card._gin_value_dict[card.number]

        remaining_2 = player.hand
        remaining_2, sets2 = player.get_sets()
        remaining_2, runs2 = player.get_runs(remaining_2)

        remaining_card_value_2 = 0
        for card in remaining_2:
            remaining_card_value_2 += card._gin_value_dict[card.number]

        if remaining_card_value_1 <= remaining_card_value_2:
            return (remaining_1, _flatten(runs1 + sets1))
        else:
            return (remaining_2, _flatten(runs2 + sets2))

    @staticmethod
    def draw(player, game):
        # strategy to just always draw the face down card
        drawn_card = game.deck.cards.pop(0)
        player.hand.append(drawn_card)

    @staticmethod
    def discard(self, player, game):
        # strategy to discard the highest value card not
        # part of a set or a run
```

```
        # NOTE: This is a strategy that could be improved.
        # What if the highest value card is a king of spades,
        # and we also have another remaining card that is the
        # king of clubs?

        # NOTE: Another way to improve things would be using "deque"
        # https://docs.python.org/3/library/collections.html#collections.deque
        # prepending to a list is not efficient.
        remaining_cards, complete_cards = self.get_best_hand(player)
        remaining_cards = sorted(remaining_cards, reverse=True)

        to_discard = remaining_cards.pop(0)
        game.discard_pile.insert(0, to_discard)

        # remove from the player's hand
        for idx, card in enumerate(player.hand):
            if (card._value_dict[card.number], card.suit) == (to_discard.
↪_value_dict[to_discard.number], to_discard.suit):
                player.hand.pop(idx)

    @staticmethod
    def can_end_game(player):
        """
        The rules of gin (our version) state that in order to end the game
        the value of the non-set, non-run cards must be at most 10.
        """
        remaining_cards, _ = player.get_best_hand()

        remaining_value = 0
        for card in remaining_cards:
            remaining_value += card._gin_value_dict[card.number]

        return remaining_value <= 10

    @staticmethod
    def should_end_game(player):
        """
        Let's say our strategy is to knock as soon as possible.

        NOTE: Maybe a better strategy would be to knock as soon as
        possible if only so many turns have occurred?
        """

        if player.can_end_game():
            return True
        else:
            return False
```

```
    def make_move(self, player, game):
        """
        A move always consistents of the same operations.
        A players draws, discards, decides whether or not
        to end the game.

        This function returns two values. The first is a
        boolean value that says whether or not the game
        should be ended. The second is the player object
        of the individual playing the game. If the player
        is not ending the game, the player returned is the
        player whose turn it is now.
        """
        # first, we must draw a card
        self.draw(player, game)

        # then, we should discard
        self.discard(self, player, game)

        # next, we should see if we should end the game
        if player.should_end_game():
            # then, we end the game
            return True, player
        else:
            # otherwise, return the player with the next turn
            return False, (set(game.get_players()) - set((player,))).pop()


class Scorecard:
    def __init__(self, player1, player2):
        self.player1 = player1
        self.player2 = player2
        self.score = pd.DataFrame(data={"winner": [], f"points": []})

    def __str__(self):
        return f'{self.score.groupby("winner").sum()}'

    def stats(self):
        pass


class Game:
    def __init__(self, scorecard, deck, ruleset, player1, player2):
        self.scorecard = scorecard
        self.deck = deck
        self.discard_pile = []
```

```
        self.ruleset = ruleset
        self.player1 = player1
        self.player2 = player2

        # shuffle deck
        random.shuffle(self.deck)

    def get_players(self):
        return (self.player1, self.player2,)

    def play(self):
        """
        Play the game until a player ends the game.
        """
        # deal cards according to ruleset
        self.ruleset.deal(self)

        # first_move should bring the game's state
        # to a consistent state.

        # Example 1: use the rule where the most
        # recent loser deals 11 cards to the other player
        # and the other player begins by discarding 1 card

        # Example 2: use another variant of the "normal" rule where each player
        # is dealt 10 cards and then the remaining cards are
        # placed face down and the first card is flipped up
        # into the discard pile. A player is chosen at random
        # and they can start the game by drawing and then discarding
        end_game, player = self.ruleset.first_move(self)

        if end_game:
            self.end_game(player)

        while not end_game:
            if len(self.deck.cards) <= 2:
                # reset game in draw
                self.reset_game()

            end_game, player = player.make_move(self)

        self.end_game(player)


    def end_game(self, game_ender):
        """
        Ending a game involves the following process:
```

```
        1. If the player ending the game if "going gin", that player
        gets 25 points plus the value of the other players remaining
        cards.
        2. The other player can add their remaining cards to any of the game␣
↪ender's sets or runs.
        3. Now, the value of the remaining cards for the player
        ending the game are compared to those of the other player,
        after the other player has potentially reduced their remaining
        cards in step 2.
        4. If the player ending the game has strictly fewer points,
        the player ending the game receives the difference between
        their remaining cards and the other players remaining cards.
        5. If the player ending the game has equal to or more points,
        the player ending the game has been undercut. The other player
        receives 25 points plus the difference between their remaining
        cards and the other players remaining cards.
        """

        def _flatten(t):
            return [item for sublist in t for item in sublist]

        def _get_rid_of_deadwood(game_ender, other_player):
            remaining_cards, complete_cards = game_ender.get_best_hand()
            other_remaining, other_complete = other_player.get_best_hand()

            combined_remaining1 = other_remaining + complete_cards
            combined_remaining1, runs1 = other_player.
↪get_runs(combined_remaining1)
            combined_remaining1, sets1 = other_player.
↪get_sets(combined_remaining1)

            combined_remaining2 = other_remaining + complete_cards
            combined_remaining2, runs2 = other_player.
↪get_runs(combined_remaining2)
            combined_remaining2, sets2 = other_player.
↪get_sets(combined_remaining2)

            remaining_card_value_1 = 0
            for card in combined_remaining1:
                remaining_card_value_1 += card._gin_value_dict[card.number]

            remaining_card_value_2 = 0
            for card in combined_remaining2:
                remaining_card_value_2 += card._gin_value_dict[card.number]

            if remaining_card_value_1 <= remaining_card_value_2:
```

```
                # remove the cards used in a set or run from other_remaining
                melds = [(x._value_dict[x.number], x.suit) for x in␣
↪_flatten(runs1) + _flatten(sets1)]
                updated_other_remaining = list(filter(lambda x: (x.
↪_value_dict[x.number], x.suit) not in melds, other_remaining))
                return updated_other_remaining
            else:
                melds = [(x._value_dict[x.number], x.suit) for x in␣
↪_flatten(runs1) + _flatten(sets1)]
                updated_other_remaining = list(filter(lambda x: (x.
↪_value_dict[x.number], x.suit) not in melds, other_remaining))
                return updated_other_remaining

        # get the "other player"
        other_player = (set(self.get_players()) - set((game_ender,))).pop()

        # get both players best hands
        remaining_cards, complete_cards = game_ender.get_best_hand()
        other_remaining, other_complete = other_player.get_best_hand()

        # is the game ender "going gin"?
        if not remaining_cards:
            winner = game_ender
            points = 25
            for card in other_remaining:
                points += card._gin_value_dict[card.number]

        else:
            # let the other_player play any deadwood/remaining cards
            # they have on the game ender's sets/runs
            other_remaining = _get_rid_of_deadwood(game_ender, other_player)

            # compare deadwood
            enders_deadwood = 0
            for card in remaining_cards:
                enders_deadwood += card._gin_value_dict[card.number]

            other_deadwood = 0
            for card in other_remaining:
                other_deadwood += card._gin_value_dict[card.number]

            if enders_deadwood < other_deadwood:
                winner = game_ender
                points = other_deadwood - enders_deadwood
            else:
                winner = other_player
                points = 25 + (enders_deadwood - other_deadwood)
```

```
        # tally score
        self.scorecard.score = self.scorecard.score.append({"winner":␣
↪str(winner), "points": points}, ignore_index=True)

        # get a fresh shuffled deck and clear out hands
        self.reset_game()

    def reset_game(self):
        # get a fresh shuffled deck and clear out hands
        self.deck = Deck()
        self.discard_pile = []
        self.player1.hand = []
        self.player2.hand = []
```

[3]:
```
deck = Deck()
strategy = Strategy()
player1 = Player('Eric', strategy)
player2 = Player('Bill', strategy)
ruleset = Ruleset()
scorecard = Scorecard(player1, player2)
game = Game(scorecard, deck, ruleset, player1, player2)
```

[4]:
```
game.play()
```

[5]:
```
print(scorecard)
```

```
        points
winner
Bill       4.0
```

[6]:
```
game.play()
```

[7]:
```
print(scorecard)
```

```
        points
winner
Bill       4.0
Eric      25.0
```

[8]:
```
game.play()
```

[9]:
```
print(scorecard)
```

```
        points
winner
Bill       4.0
Eric      50.0
```

Yes. It works the way it should be.

1.4 Question 4

[10]:
```
def game_over(scorecard):
    winning_scoreboard = scorecard.score.groupby("winner").sum().reset_index().
↪loc[scorecard.score.groupby("winner").sum().reset_index()['points'] >= 100.
↪0, :]
    return winning_scoreboard['winner'], winning_scoreboard.shape[0] > 0.0
```

[11]:
```
Eric_points = 0
Bill_points = 0
```

[12]:
```
while not (Eric_points >= 3) and not (Bill_points >= 3):
    deck = Deck()
    strategy = Strategy()
    player1 = Player('Eric', strategy)
    player2 = Player('Bill', strategy)
    ruleset = Ruleset()
    scorecard = Scorecard(player1, player2)
    game = Game(scorecard, deck, ruleset, player1, player2)

    game_done = False
    while not game_done:
        game.play()
        print(scorecard)
        winner, game_done = game_over(scorecard)
    if winner.iloc[0] == 'Eric':
        Eric_points += 1
    else:
        Bill_points += 1
print(f""" final score:
Eric: {Eric_points}
Bill: {Bill_points} """)
```

```
        points
winner
Bill      21.0
        points
winner
Bill      46.0
        points
winner
Bill      71.0
        points
winner
Bill      96.0
        points
```

```
winner
Bill     121.0
        points
winner
Bill      23.0
        points
winner
Bill      23.0
Eric      25.0
        points
winner
Bill      48.0
Eric      25.0
        points
winner
Bill      48.0
Eric      50.0
        points
winner
Bill      73.0
Eric      50.0
        points
winner
Bill      73.0
Eric      75.0
        points
winner
Bill      98.0
Eric      75.0
        points
winner
Bill      98.0
Eric     100.0
        points
winner
Bill       8.0
        points
winner
Bill       8.0
Eric      25.0
        points
winner
Bill      33.0
Eric      25.0
        points
winner
Bill      33.0
Eric      50.0
```

```
        points
winner
Bill      33.0
Eric      75.0
        points
winner
Bill      33.0
Eric     100.0
        points
winner
Eric      27.0
        points
winner
Eric      52.0
        points
winner
Bill      25.0
Eric      52.0
        points
winner
Bill      50.0
Eric      52.0
        points
winner
Bill      75.0
Eric      52.0
        points
winner
Bill     100.0
Eric      52.0
        points
winner
Bill      23.0
        points
winner
Bill      23.0
Eric      25.0
        points
winner
Bill      23.0
Eric      50.0
        points
winner
Bill      23.0
Eric      75.0
        points
winner
Bill      48.0
```

```
Eric      75.0
        points
winner
Bill      73.0
Eric      75.0
        points
winner
Bill      98.0
Eric      75.0
        points
winner
Bill     123.0
Eric      75.0
 final score:
Eric: 2
Bill: 3
```

As above, final score is printed.

1.5 Question 5

[14]:
```python
class myStrategy:

    @staticmethod
    def get_best_hand(player):

        def _flatten(t):
            return [item for sublist in t for item in sublist]

        # this strategy is to get the runs then sets in that order,
        # count the remaining card values, then reverse the process,
        # get the sets then runs in that order, then count remaining
        # card values
        remaining_1 = player.hand
        remaining_1, runs1 = player.get_runs()
        remaining_1, sets1 = player.get_sets(remaining_1)

        remaining_card_value_1 = 0
        for card in remaining_1:
            remaining_card_value_1 += card._gin_value_dict[card.number]

        remaining_2 = player.hand
        remaining_2, sets2 = player.get_sets()
        remaining_2, runs2 = player.get_runs(remaining_2)

        remaining_card_value_2 = 0
        for card in remaining_2:
            remaining_card_value_2 += card._gin_value_dict[card.number]
```

```
        if remaining_card_value_1 <= remaining_card_value_2:
            return (remaining_1, _flatten(runs1 + sets1))
        else:
            return (remaining_2, _flatten(runs2 + sets2))

    @staticmethod
    def draw(player, game):
        # strategy to just always draw the face down card
        drawn_card = game.deck.cards.pop(0)
        player.hand.append(drawn_card)

    @staticmethod
    def discard(self, player, game):
        global to_discard
        def flatten(x):
            return [item for sublist in x for item in sublist]
        # strategy to discard the highest value card not
        # part of a set or a run

        # NOTE: This is a strategy that could be improved.
        # What if the highest value card is a king of spades,
        # and we also have another remaining card that is the
        # king of clubs?

        # NOTE: Another way to improve things would be using "deque"
        # https://docs.python.org/3/library/collections.html#collections.deque
        # prepending to a list is not efficient.
        remaining_cards, complete_cards = self.get_best_hand(player)
        partial = []
        for key, value in Counter(remaining_cards).items():
            if value == 2:
                partial.append([card for card in remaining_cards if card ==␣
↪key])
        partial = flatten(partial)
        remaining_cards = sorted(remaining_cards, reverse=True)
        for i, card in enumerate(remaining_cards):
            if card not in partial:
                to_discard = remaining_cards.pop(i)
                break

        game.discard_pile.insert(0, to_discard)

        # remove from the player's hand
        for idx, card in enumerate(player.hand):
            if (card._value_dict[card.number], card.suit) == (to_discard.
↪_value_dict[to_discard.number], to_discard.suit):
```

```
                player.hand.pop(idx)

    @staticmethod
    def can_end_game(player):
        """
        The rules of gin (our version) state that in order to end the game
        the value of the non-set, non-run cards must be at most 10.
        """
        remaining_cards, _ = player.get_best_hand()

        remaining_value = 0
        for card in remaining_cards:
            remaining_value += card._gin_value_dict[card.number]

        return remaining_value <= 10

    @staticmethod
    def should_end_game(player):
        """
        Let's say our strategy is to knock as soon as possible.

        NOTE: Maybe a better strategy would be to knock as soon as
        possible if only so many turns have occurred?
        """

        if player.can_end_game():
            return True
        else:
            return False

    def make_move(self, player, game):
        """
        A move always consistents of the same operations.
        A players draws, discards, decides whether or not
        to end the game.

        This function returns two values. The first is a
        boolean value that says whether or not the game
        should be ended. The second is the player object
        of the individual playing the game. If the player
        is not ending the game, the player returned is the
        player whose turn it is now.
        """
        # first, we must draw a card
        self.draw(player, game)

        # then, we should discard
```

```
        self.discard(self, player, game)

        # next, we should see if we should end the game
        if player.should_end_game():
            # then, we end the game
            return True, player
        else:
            # otherwise, return the player with the next turn
            return False, (set(game.get_players()) - set((player,))).pop()
```

[27]:
```
deck = Deck()
strategy = Strategy()
mystrategy = myStrategy()
player1 = Player('Eric', strategy)
player2 = Player('Bill', mystrategy)
ruleset = Ruleset()
scorecard = Scorecard(player1, player2)
game = Game(scorecard, deck, ruleset, player1, player2)
```

[28]:
```
Eric_points = 0
Bill_points = 0
```

[29]:
```
while not (Eric_points >= 3) and not (Bill_points >= 3):
    deck = Deck()
    strategy = Strategy()
    mystrategy = myStrategy()
    player1 = Player('Eric', strategy)
    player2 = Player('Bill', mystrategy)
    ruleset = Ruleset()
    scorecard = Scorecard(player1, player2)
    game = Game(scorecard, deck, ruleset, player1, player2)

    game_done = False
    while not game_done:
        game.play()
        print(scorecard)
        winner, game_done = game_over(scorecard)
    if winner.iloc[0] == 'Eric':
        Eric_points += 1
    else:
        Bill_points += 1
print(f""" final score:
Eric: {Eric_points}
Bill: {Bill_points} """)
```

```
        points
winner
```

```
Eric      10.0
        points
winner
Eric      35.0
        points
winner
Eric      60.0
        points
winner
Bill      25.0
Eric      60.0
        points
winner
Bill      25.0
Eric      85.0
        points
winner
Bill      25.0
Eric     110.0
        points
winner
Eric      10.0
        points
winner
Bill      25.0
Eric      10.0
        points
winner
Bill      25.0
Eric      35.0
        points
winner
Bill      25.0
Eric      60.0
        points
winner
Bill      50.0
Eric      60.0
        points
winner
Bill      50.0
Eric      85.0
        points
winner
Bill      50.0
Eric     110.0
        points
winner
```

```
Bill      13.0
        points
winner
Bill      38.0
        points
winner
Bill      38.0
Eric      25.0
        points
winner
Bill      38.0
Eric      50.0
        points
winner
Bill      63.0
Eric      50.0
        points
winner
Bill      88.0
Eric      50.0
        points
winner
Bill     113.0
Eric      50.0
        points
winner
Bill      25.0
        points
winner
Bill      50.0
        points
winner
Bill      50.0
Eric      25.0
        points
winner
Bill      75.0
Eric      25.0
        points
winner
Bill     100.0
Eric      25.0
        points
winner
Eric      25.0
        points
winner
Bill      25.0
```

```
Eric      25.0
        points
winner
Bill      50.0
Eric      25.0
        points
winner
Bill      75.0
Eric      25.0
        points
winner
Bill     100.0
Eric      25.0
 final score:
Eric: 2
Bill: 3
```

I modified the discard method. Bill wins with new strategy

1.6 Pledge

By submitting this work I hereby pledge that this is my own, personal work. I've acknowledged in the designated place at the top of this file all sources that I used to complete said work, including but not limited to: online resources, books, and electronic communications. I've noted all collaboration with fellow students and/or TA's. I did not copy or plagiarize another's work.

> As a Boilermaker pursuing academic excellence, I pledge to be honest and true in all that I do. Accountable together – We are Purdue.