Analysis Report: President (Card Game)

Alex Chan | 08 Nov 2020 | 1405Z

Note: This report isn't split by the five points we need to cover; rather, it is split into the various files used in my program, each corresponding to a different subproblem that was solved. Each file section is further split into the most important individual functions and classes, which is where the bulk of the analysis (complexity analysis/computational thinking/tradeoffs/extensions) will be discussed for each.

Problem Description

My project revolves around implementing the card game **President** (Basic Explanation: https://www.pagat.com/climbing/president.html) for users to play, and developing an AI algorithm that can fill in for a player. There are three overarching subproblems that I have solved in order to create the program: have a system to deal with all the cards, create the base game functionality with a user interface in terminal, and finally create an AI that can play in place of a player.

The basic idea around this game is that cards are dealt to players, and players go around, playing their cards. How they play their card depends on the card that came before - if it was a single card, then they'd have to play a higher one. If it was a poker hand, then there are other rules that determine if they can play a particular hand. The game continues until all players have played all their cards, and the order that they do determines their role in the next round. Depending on the player's role, they will either have the chance to swap a select number of cards of their choosing with another player, or be forced to give up what the other party requests.

Program Entry Point: `main.py`

This serves as the entry point for the program. Basically, it uses the functionality from `base_game.py` to perform the higher-level actions necessary to run the program.

Base Game: `base_game.py`

Global Variables

- This file has three primary global variables:
    - `total_rounds`, an integer which specifies the total number of rounds to be played for the game
    - `num_players`, an integer which specifies the number of players in the game
    - `players`, a list which contains all the `Player` objects in the game

`setup()`

Sets up the game by allowing user to specify the number of rounds, players, and their names and AI status

- Using **decomposition** and **abstraction** (by outlining the high-level requirements for this function), I determined that this function would have several main components:
  - Print a welcome message to the user
  - Allow the user to choose the number of players and number of rounds
  - For each player, allow the user to input their name and whether or not they are an AI
- Using **pattern recognition** and **algorithm design**, I determined the following:
  - We could iterate one time for each player, and allow the user to input that player's data
  - So, I implemented an algorithm along the lines of:

    ```
    Set counter = 1
    Iterate for each player:
      Ask user to input player name and store it
      Ask user whether or not player is AI and store response

      If player is AI, add a new AI to the players list
      Otherwise, add a new Person to the players list
    ```

- **Extensions**:
  - I could potentially add more options for the user to choose from, like which moves are valid, who can pass, etc. (this should be quite easy, since most of the functions in the game have arguments that can control various things - it would just be a matter of calling functions with different arguments)

`deal_to()`

Shuffle a full deck of cards and deal them out to a specified number of players - using my algorithm, each player will receive the exact same number of cards (and so this function also serves to return the cards leftover from the deck)

- **Algorithm design:**

  ```
  Function deal_to(players):
    Set deck to the full 52-card deck of cards
    Set cards_left to the length of deck (52)
    Set leftover to cards_left % number of players
    Set keep_dealing to True

    While keep_dealing is true:
      Iterate for each player:
  ```

```
    If cards_left <= leftover,
      Set keep_dealing to false
      Break out of loop
    Otherwise,
      Add to player's hand a random card from the deck
      Remove that card from the deck
      Reduce cards_left by 1

  Return deck, which contains all the leftover cards
```

`get_lowest_card()`

This returns the lowest card that any player could potentially have, given the cards leftover from dealing to players (typically this would be the three of diamonds, but if there are cards leftover, that card may not have been dealt)

- Using **abstraction**, I realized that this could use **recursion** since there are two cases:
    - The base case is that the lowest possible card is not in the leftover card deck, in which case the lowest card the players have *would have to be* that lowest possible card
    - The recursive case is if we find that lowest possible card in the leftover - we can apply recursion by returning `get_lowest_card()` and giving it the leftover deck without the lowest possible card, and the next lowest card as its lowest possible card - so, it would call the function again but now checking for the next possible card (e.g. for the 3 of diamonds, it would check for the 3 of clovers, then the 3 of hearts, and so on)

`do_round()`

Creates a round system that initializes who goes first, in what order, and what (if any) trading needs to take place. It should then allow each player to go in turn, and finally end the round once all players have finished.

- I used **abstraction** and **decomposition** again to break down the problem into 3 main subproblems, as well as **algorithm design** to create this high-level algorithm:
    - **Setup** deals with all the initial things that need to dealt with before starting the actual round, like:
        - Dealing all the cards
        - Figuring out who goes first (first round, this is whoever has the lowest card; afterwards, this is the president)
        - Starts the trade for any subsequent round
        - Outputs to the user the round information and the order of the players
        - Initializes several variables:
            - `round_ended = False` is a boolean that is indicative of whether the round has

        ended
        - `is_first_turn = True` is a boolean that is indicative of whether this is the first turn around (we need to know this to have the correct player go first initially - all subsequent turns just follow the players list order)
        - `prev_move = '*'` is typically a `Move` object which is used to figure out what valid moves a player has when they have their turn afterwards - however, it can also be the string `'*'` which indicates there is no previous move (either no players have gone, or everyone else has passed)
        - `num_passes = 0` is an integer that increments whenever someone passes, and is used to allow the player who played initially to play any card they want
        - `players_finished = 0` is an integer that is used to tell when the round should end
        - `moves = []` is a list which contains all the moves that players play (it is a list, since we want the moves in order of how they were played)
    - **Turns** iterates through each player using a while loop (`while not round_ended`), and allows them to play cards until they have all (minus 1 player) have finished:
        - For the first turn around (to start with the previously determined first player) or any subsequent round (to skip finished players), it skips that iteration of the loop using the following conditional:

          ```
          If current player is finished or (is_first_turn and current index is
          not starting_player_index), skip the rest of this iteration
          ```

        - Next, if the current player has not been skipped according to the above conditional, it prints out the player information (hand, name), as well as the previous move
        - Then, it allows the player or AI to play their move, depending on various boolean values:
            - If this is the first turn and this is the first round, set `move = player.do_move(prev_move, lowest_card)` -- we call it with `'*'` to show there is no previous move, and `lowest_card` to indicate it must contain the lowest card, since it's the very first round
            - Otherwise, set `move = player.do_move(prev_move)`
            - For either of the above, set `prev_move = move` and `is_first_turn = False` for the next player
            - If this is not the next turn, set `move = player.do_move(prev_move)`
                - If `move` becomes `'*'`, we know the player passed - in this case, increase `num_passes` by 1
                    - If `num_passes` is equal to `num_players - 1`, this means all players passed on the next player's move, which means we can set `prev_move` to `'*'`, which means the next player can play any move in their hand

            - Otherwise, reset `num_passes` to 0, and set `prev_move` to `move`
        - After that, we check if the current player has just finished now:
            - If so:
                - We print it out to the user
                - Increase `players_finished` by 1
                - Add place finished to the player's finishing record
                - If `players_finished` is equal to `num_players - 1`, this means the current player is the final player (other than one other) to finish, so we can set `round_ended = True` and break out of the loop
            - Otherwise, we simply print the card played to the player, and move on to the next player
        - Finally, we add the current move played to the `moves` list
    - **Ending of the round**:
        - For that last player who didn't finish, we add last place to their finishing record
        - We then use `get_finishing_roles()` to get all their finishing roles
        - We then print out all the roles of the players
        - Then, we reset every player (get rid of their cards and reset their finished status)

Extensions

- I would prefer to further use abstraction and decomposition to break this down into smaller functions, so it's a bit more manageable

`get_finishing_roles()`

Assigns each player a role based on their finishing position.

- I quite enjoyed approaching this problem, since it involved some thinking to get it to work without simply using brute force. I made a kind-of table to give me an idea of which roles should be assigned depending on how many players are in the game:

```
{p: 'President', vp: 'Vice-President', n: 'Neutral', vb: 'Vice-Bum', b: 'Bum'}
1 2 p b [0]
1 2 3 p n b [0, 1]
1 2 3 4 p vp vb b [0, 1]
1 2 3 4 5 p vp n vb b [0, 1, 2]
1 2 3 4 5 6 p vp n n vb b [0, 1, 2]
1 2 3 4 5 6 7 p vp n n n vb b [0, 1, 2, 3]
```

- I decided it would make the most sense to somehow iterate using a loop, but where the roles for the players are filled in inwards (so president-bum first, then vices next, then all the neutrals in the center)

- Using **algorithm design**, I came up with this algorithm (you can see the explanations in the comments):

```
Sort players (from highest role to lowest)
Set roles to a list: ['President', 'Vice-President', 'Neutral', 'Vice-Bum',
'Bum'}

Loop counter from 0 to ceiling of (num_players / 2):
  // This is what is represented in the [0, 1, 2] above - the range it
iterates through
  If counter >= 2
  // Any player which is 2 or more spaces in should be assigned 'neutral'
  Or counter is num_players - 1 - counter {
  // This is if 3 players, in which case the middle is neutral too
    players[counter] and players[-counter - 1] are assigned roles of roles[2]
    // Sets role of player at index counter,
    // or player which is counter indices from the end, to neutral
    Continue to next iteration
  }

  Otherwise,
    Set players[counter] role to roles[counter]
    Set players[-counter - 1] role to roles[-counter - 1]
    // Sets the outermost two players (first and last two) to the first and
last two
    // roles within the roles list
```

`do_trade()`

Creates a trading system before the round starts for the various roles

- Using **abstraction** and **pattern recognition**, I noticed how this subproblem was very similar to the previous subproblem (`get_finishing_roles`), because again I would need to deal with players by working from the outside (president and bum) to the inside (neutrals), in order to facilitate trade
- So, I used a very similar approach, by iterating from the outside to the inside by iterating to the ceiling of `num_players / 2`, and accessing the outermost players using `players[counter]` and `players[-counter - 1]`
- Using **decomposition**, I further broke this problem down by creating another function, `trade_between()` to actually facilitate the trade once this function figured out which two players would trade (and which would have the trading power)

`trade_between()`

Allows players to trade based on various things:

- It accepts two arguments, both being `Player` objects - one upper player (president/vice pres), and one lower (vice bum/bum)
- It determines how many cards to trade by accessing the upper player's latest finishing place (if it's first, indicating president, then they trade 2; if second, indicating vices, they trade only 1)
    - This is calculated using `num_cards = 3 - upper_player.finishing_record[-1]`

Cards, Hands, and Move Functionality: `card.py`

`Card` class

Creates card objects with a particular value and suit, and prints them out nicely

- Using OOP, I managed to create this card class which really serves as the backbone for the rest of the game, since it is this class that creates all the card objects and allows us to manipulate/compare/etc. the cards
- There are many variants of the value - first, initialization takes in a value from 1 to 13, where 1 = A, 11 = J, 12 = Q, 13 = K
    - However, we need an integer value that can be used to compare cards, so cards with values 1 and 2 have values stored that are increased by 13 (A = 14, 2 = 15)
    - We also need a display value, so all the letter cards also have a display value (A, J, Q, K)
- For the suits, their information is stored as class variables:
    - Each suit is stored as a dictionary, with a 'value' key (from 0 to 3, which is used to convert number during initialization to a suit), a 'symbol' key (which stores the unicode symbol for the suit), and a 'colour' key for which colour the card of this suit should be outputted as (dictionary is very intuitive, makes it easy to access information, and allows for quick searching)
    - All the suit dictionaries are then stored in a list in order from lowest to highest (list maintains this order nicely)
- In addition, there are the `__eq__` and `__lt__` comparison methods, which first compare the value, and then compare the suits of the card and another card to determine which is greater/lesser/equal

`Hand` class

`Hand.subtract()`

Takes another variable, which contains a combination of cards/moves/hands, and removes all the cards in that variable from this class

- We use **recursion** for this function, very similarly to the box problem on the exam, since it can contain any combination/branching of cards, inside hands, moves, and lists - we only care about the actual cards within the other variable
- The **runtime complexity** is **O(n)**, where n is the total number of cards + the total number of moves/lists/hands (including within all moves/lists/hands inside):
    - It needs to loop through each element in the other variable
        - If that element is a card, we remove the card from this hand
        - If that element is a list/move/hand, we call `Hand.subtract()` recursively for that list/move/hand, and so on in order to remove all the cards
    - Thus, it needs to iterate for every card, move, hand, and list within the other variable

`Hand.get_valid_moves()`

Can take a hand of cards, and a previous move, and return all valid moves

- Throughout this section, runtime complexity analysis will be included in square brackets []:
    - Assume that n is the total number of cards in the hand
    - Assume that m is the total number of moves in the hand
- Using **abstraction** and **decomposition**, I broke down this function into some main components:
    1. Set the initial local variables (has_previous_move, first_move, option) [**O(1)** as it is just assigning values]
    2. Get and store `value_frequencies` (a dictionary with card values as keys, and the lists of cards with that value as values) [**O(n)**, as we iterate through each card]
    3. Check if the hand is less or equal to the previous move's size and returns accordingly [**O(1)** as it is just conditionals]
    4. Loops through each helper function to check for one-card moves, two-to-four-card moves, and five-card combos - if there is a previous move, returns immediately
    5. Otherwise, returns the list of valid moves, but sorted by hand type [**O(m log m)**, according to the `sorted_by_hand_type()` runtime analysis]
- I'll focus more on the runtime complexity and algorithm of step 4, since it's the most complex and has the most substance - however, the overall runtime complexity is beyond the scope of this course, as it involves big-O runtime complexities involving combinatorics:
    - First, we create a dictionary of the various helper functions, where:
        - The *key* is the helper function
        - The *value* is a dictionary containing a 'range' key (which has a value of either an integer [a == size], or a 2-element array [a <= size <= b] which specifies what the size of the previous move must be), and an 'args' key (which has a list value which contains all the arguments needed for that helper function)

- Then, we loop through the helper functions dictionary [does not affect runtime, since it loops up to 3 times], and:
  - If there is a previous move, check if the previous move's size is within the helper function's range - only if not, skip the rest of the iteration
  - Call the helper function with the 'args' specified in the dictionary [See below for runtime]
  - If there is a previous move, immediately return all valid moves (do not iterate to the next helper function) [**O(m log m)**, according to the `merge_sort()` runtime analysis]
- If no previous move, then loop through the rest of the helper functions, and:
  - If this is the first move, then only include the moves that contain the lowest card
  - Otherwise, include all moves [**O(m log m)**, according to the `sorted_by_hand_type()` runtime analysis]
- Return all the moves, sorted by hand types

- Now, there are several helper functions:
  - For a previous move of size 1 (or no previous move):
    - Loop through all cards in the hand, and adds every single one individually, as a move, to the list of valid moves [**O(n)**, as we iterate through each card]
  - For a previous move of size 2-4 (or no previous move):
    - If there is a previous move, set `move_sizes` to a one-element array containing the previous move's size
    - Otherwise, set `move_sizes` to `[2, 3, 4]` (for each possible size)
    - Set `move_type_dict = {}` (empty dictionary)
    - Loop through each move size in move_sizes [No effect on runtime, since it is just a constant of 1, 2, or 3]:
      - Loop through each card value in `value_frequencies` [**O(v)**, where v is the distinct card values of the cards in the hand]:
        - If the number of cards with that card value is less the the current move size, skip the rest of the iteration
        - Set combinations to the list created from `itertools.combinations()` from the list of cards with this card value, where each combination has move size number of cards [**O(r(^nC_r)**, according to the documentation of combinations, where r is the move size]
        - Loop through each combination [**O((^nC_r)**, since that is the mathematical value for the number of combinations from a list of n cards where each combination is 5 long]:
          - Add each combination to valid moves
  - For a previous move of size 5 (or no previous move):
    - Set `move_type_dict = {}` (empty dictionary)

        - Set combinations to the list created from `itertools.combinations()` from the list of cards with this card value, where each combination has 5 cards [**O(5($^nC_5$)**, according to the documentation of combinations]
        - Loop through each combination [**O(($^nC_5$)**, since that is the mathematical value for the number of combinations from a list of n cards where each combination is 5 long]:
            - If the combination is a valid move according to `get_type()`, then add it to valid moves

Tradeoffs

- I thought quite a lot about how to deal with a hand's number of valid moves - currently, we are prioritizing memory over runtime since we never store the valid moves of a hand as an attribute - rather, it can only be accessed by running `get_valid_moves()` each time, so it needs to run every time
    - This was chosen because hands almost always change, so it would probably be even more inefficient to store valid moves as a variable
    - If we were to access the valid moves as a variable, then we would need to obtain the valid moves of the most recent hand - this means that *every single time* the hand changes (i.e. appending, extending, removing, etc.), then the program needs to also evaluate `get_valid_moves()` again and again
    - So, it is most ideal to evaluate and return `get_valid_moves()` each time it needs to be accessed, rather than storing it in memory (as in effect, even though that uses more memory, it actually also uses more runtime for each change in the hand)

`Move` class

Hand object that has added functionality of being a playable move by players

- This class includes methods that can compare two moves and see which one is higher
    - By implementing these comparison methods, we can both use the built-in `sort()` function in Python, but also compare moves simply using comparison operators like `>` and `<` to implement our own functions (like `sorted_by_hand_type()`) below

`Move.get_type()`

- Using **computational thinking**, I basically broke this function down into many conditionals and developed algorithms for each particular case - I'll provide a relatively high-level algorithm below according to the code:
    - First, we need to sort the move from least to greatest card
    - If move size is 0, `hand_type` is `'empty'` [**O(1)** runtime, since it's just a conditional]
    - If move size is 1-4: check if all cards are the same (for either a one-card, pair, three-of-a-kind, or four-of-a-kind)

        - Set `first_card` to the first card
        - Loop through each card in the move, and if the card's value is not equal to that of `first_card`, `hand_type` is `'scattered'` [**O(n)** runtime, where n is the number of cards, since it must loop through each card]
        - Otherwise, if all cards are of the same value, assign `hand_type` to the according hand type (e.g. 1 card = one-card, 2 cards = pair, etc.)
    - If move size is 5: check for 5-card combos [**O(n)** runtime, where n is the number of cards, since it must loop through each card]:
        - Set `first_card` to the first card (lowest, since cards are sorted)
        - Set booleans `is_straight` and `is_flush` both to true
        - Create empty list called `repeat_value_counter` with elements that contain how often values repeat (e.g. `[2, 3]` means one value has 2 cards, the other has 3 cards)
        - Set integer `repeat_values` to 0
        - Loop through each card in the move:
            - If `is_flush` is still true, but the current card's suit is not equal to the first card's suit, set it to false
            - If `is_straight` is still true, but the current card's value is not equal to the previous card's value + 1, set it to false
            - On the first card, set `repeat_values` to 1
            - Otherwise, if the current card's value is not equal to the previous card's value, increase `repeat_values` by 1
            - Otherwise, append the current `repeat_values` to `repeat_value_counter`, and reset `repeat_values` to 1
        - Finally, we check through all the conditionals to determine which hand type the move is:
            - If `is_flush` and `is_straight`, `hand_type` is `'straight flush'`
            - If `is_flush` and not `is_straight`, `hand_type` is `'flush'`
            - If not `is_flush` and `is_straight`, `hand_type` is `'straight'`
            - If neither, and `repeat_value_counter` contains only 2 and 3, `hand_type` is `'full house'`
            - If neither, and `repeat_value_counter` contains only 1 and 4, `hand_type` is `'four-of-a-kind plus one'`
            - Otherwise, `hand_type` is `scattered`
    - If move size is none of the above, `hand_type` is `'scattered'` [**O(1)** runtime, since it's just a conditional]
- Overall, this function has a worst-case runtime of O(n), where n is the number of cards in the move

`sorted_by_hand_type()`

Sort hand of cards, but with the various hand types sorted within themselves

- This function expands the functionality of sort, by sorting a particular block of moves with a particular hand type, and then sorting those blocks separately - this is if you don't want all the moves in order, but rather all the moves with hand types in order, but sorted in reverse within those hand types
- Using **computational thinking**, I figured out how to split this problem effectively and create a simple algorithm:
    - First, I create a dictionary of all the moves, where the moves are split into various hand types (keys of dictionary), and every value is a list holding all the moves that belong to that hand type. We do this by iterating through each move:
        - If the move hand type is in the dictionary, then append the move to that hand type's list
        - Otherwise, create a new key according to the move hand type, and create a new list as the value containing just this move
    - Then, using merge sort, we iterate through each hand type in the dictionary, and sort the list of all the moves that fall under that hand type
    - Finally, we sort the hand types in the dictionary, and return a final list by extending all the lists in the dictionary which are already sorted
        - However, note that since dictionary loses the ability to maintain order, I created a list containing all the keys in the dictionary, and then add the lists for each key in order according to the sorted order of all these keys (i.e. hand types). Then, I would extend their respective lists to the final list by accessing their lists using this sorted list of keys
- The **runtime complexity** of this function is a little bit complex:
    - First, let's assume that n is the number of total moves, and h is the number of unique hand types
    - For the first section, where each move is put into the dictionary, we get n computations (since we iterate through each move)
    - Then, for sorting the lists for each hand type, we get a runtime complexity of $n \log \frac{n}{h}$ because:
        - We need to iterate through each hand type, which is h
        - Then, we need to perform merge sort for each list of moves belonging to each hand type - on average, each hand type will have about n/h moves in each. So, since it is just merge sort, we first need to divide and merge the list for each level. This has a runtime of O(n/h) since there are twice the number of lists (x2), but half the number of items per list (/2). Additionally, there are $\log_2 \frac{n}{h}$ levels, as we are dividing the list in two each time. So, the complexity of process() overall is $\log \frac{n}{h}$ * $\frac{n}{h}$ = $\frac{n}{h} \log \frac{n}{h}$
        - Multiplying the two, we get $n \log \frac{n}{h}$

    - Then, we need to use merge sort on the hand types. Since there are h hand types, the runtime complexity for this is h log h (given the complexity of h)
    - Finally, we need to iterate for each hand type, in order to extend the list of moves for that particular hand type, which is h computations
    - Overall, we get n + $n \log \frac{n}{h}$ + h log h + h
    - Since h only goes to a maximum of 9, we can deduce that it is relatively constant compared to n, which can increase dramatically. So, since big-O notation only considers the highest-order term and ignores constants, we get a runtime complexity of **n log n**
- As a **tradeoff**, I decided to prioritize runtime over memory, since we're using merge sort here instead of quicksort, for multiple reasons:
    - Merge sort requires many copies of the lists, and does not sort the list in place - so, we're using up more memory
    - However, since the moves are certainly not in random order, given the `get_valid_moves()` function which gets all the moves of a certain type before another, it may be the case that quicksort approaches its worst-case runtime complexity of O(n^2) - for merge sort, which has a runtime complexity of O(n log n) may be better for runtime for this reason

Player Parent Class: `player.py`

Creates player objects which contain all the information of each player

- Includes methods that reset the player's hand and finished status, can give cards to other players, get the parameters required (like valid moves, if the player can pass) to determine the move, and initialize the player

Extensions

- Potentially, I could allow different players to play one common game over local connection or online - I was thinking of adding an additional attribute to each player, like a connection/IP address (I don't really know how it works yet), and then when we print, we only print it to that particular computer

Person Player: `person.py`

A child class of `Player`, which adds extra functionality/overrrides certain methods to allow for user input

- The methods here are relatively straightforward, the main idea is simply to print out all the options to the user and then allow them to choose which one to trade/play/etc.

Artificial Intelligence: `ai.py`

A child class of `Player`, which adds extra functionality/overrrides certain methods that allow the computer to make decisions

`ai_move()`

Return the AI's determined move

- Using **abstraction**, this function serves as a high-level function that calls a set of other functions (subproblems) to return the move accordingly - it doesn't do anything alone

Extensions

- I could further develop the AI algorithm to consider more things, determine the best move more efficiently, and reduce the runtime complexity - some options include:
    - Considering the behaviour of other players and adapting to their behaviours (Machine Learning, might be way too complicated though)
    - Considering high cards/unbeatable cards differently from other cards, since they need to be played sparingly according to the situation
    - Consider actually passing, which is not a possibility in this current implementation unless passing is the only option

`__ai_determine_subsequent_move()`

Considers the AI's valid moves, and which one is most beneficial to play

- Using **abstraction**, the basic idea of this function is to loop through each move in the AI's set of valid moves, and then evaluate the hand that results when that move is player - then, it compares these leftover hands and returns the greatest one based on the `card_tools.greater_than()` function
- On a very basic level, using **pattern recognition**, I know that this pattern is very similar to a simple maximize function - however, this stores and returns different things and compares different things
- The **runtime complexity** of this function is **O(n^2) + O(nc)**, assuming n is the number of valid moves of the hand since, and c is the number of cards in the hand:
    - It needs to iterate through each move, which is O(n)
    - Within each iteration, it needs to call `subtract()` and `greater_than()`
        - `subtract()` takes **O(c)** time (see its own runtime complexity)
        - `greater_than()` takes **O(n)** time (see its own runtime complexity)
    - Overall, we see that the runtime is equivalent to n(c + n) = nc + n^2
        - We cannot ignore c (and thus ignore the entirety of nc as a lower-order term) since c can also affect the runtime quite substantially depending on how many cards there are, rather than serving simply as a constant

- As a **tradeoff**, I decided to use this algorithm to determine the AI's next move instead of `move_with_minimum_moves_to_victory()` because of the insane runtime required for that function compared to this (this has runtime complexity of O(n^2) + O(nc) vs the other's O(n!))

`ai_choose_cards()`

Chooses cards from the AI's hand to give away, based on which cards are the most beneficial

- Using **pattern recognition**, I realized this was very similar to the previous function, `__ai_determine_subsequent_move()`, but instead of looping through each move in valid moves, it loops through each card and compares the hand if that card was removed
- However, one slight difference is that instead of getting the best card, it may need to get the best 2 cards
    - I solved this by instead of storing the best variables as singular Hand objects, lists, and cards, it stores them as lists of these things
    - First, it fills these lists with None, based on how many cards need to be chosen (i.e. if 2 cards, then the list has 2 None's)
    - Then, for each iteration when checking each card, it does not simply re-assign those best variables, but rather loops through each element of the best lists, and inserts it into the list at the index, and then pops the list at the end when it finds an element it is greater than
    - Using **pattern recognition**, I realized that this is basically a queue data structure!

Card Tools: `card_tools.py`

`full_deck()`

Stores the entire deck of cards, including its output format, its number, and suit

- We minimize the runtime complexity of the storage of these cards, by **trading off** space complexity, because:
    - We have a global variable called `deck`, which is assigned first an empty Hand
    - Only the first time `full_deck()` is run, does the program actually run and fill in `deck`
    - Any other time, it simply accesses the global variable `deck`
    - So, I made the program so that it only runs once, when called for the first time - however this sacrifices memory space since now that `deck` variable must store that information
    - It has a runtime of O(n), where n is the number of cards in the deck (for this, there are 52) - however, if we were to fill, and thereby loop for each call of `full_deck()`, we would have runtime of O(n) * t, where t is the number of times we access the deck - since we need to access the deck throughout runtime, it is best to sacrifice some memory for smaller runtime complexity

`greater_than()`

Return whether the first set of moves is greater than the second, based on the hand types each set of moves consists of

- This compares two lists of moves by comparing the hand types each has (regardless of how many of each hand type it contains, or which actual hand is greater - ONLY the hand types)
- The reason for this function is for the AI to determine
- Initially, I was going to use a dictionary to store all the hand types each list of moves - however, there were two main issues:
    - First of all, dictionaries do not keep order, and since the move list arguments are both already sorted from greatest to lowest, we need to keep the order in which the hand types are appended to collection
    - Second of all, dictionaries don't have the flexibility of accessing a particular index for multiple dictionaries at a time simply in one loop - with a list, we can iterate through a range from 0 to len(list) - 1 and access both lists at once
- With that out of the way, this is the **basic algorithm** I used for this function (see [] for runtime complexity analysis, where n is the total # of moves, and h is the number of hand types):
    - Loop through each list of moves, and add the move's hand type (as an index from least to greatest) to the respective hand type list IF that list is empty, or the last hand type in that last is not equal to this move's hand type [**O(n)** since we iterate through each move]
        - This is equivalent to using hash search (`in`) with a dictionary, since as all the moves are already in order, all the hand types will also be in order, so checking the last hand type is the same as searching for a hand type in the lsit of hand types [**O(1)** because of hash search]
    - Then, we loop through every hand type in the list of moves with fewer moves, and compare it with the hand type at the same index in the other list of moves [**O(h)** since we iterate through each hand type]
        - Then, if they are equal, we go to the next hand type in each list
        - Otherwise, we return true if the hand type in list1 > hand type in list2, and false otherwise
    - If all the hand types are equal, then we return true if the number of hand types in list1 > number of hand types in list2
- From above, the runtime complexity is **O(n)**, since O(h) can be ignored as practically a lower-order term, since h can only go up to a maximum of 9

`move_with_minimum_possible_moves_to_victory()` *(Possible Extension)*

Returns the minimum possible moves to victory and the move that would permit that given a hand

- This was my first attempt in my approach of figuring out a suitable AI for President - in theory, this function would use **recursion** to try every single move in a move, and each subsequent move after playing the initial moves, and so on, until all cards had been used, and return how many moves it would take
- However, the runtime complexity is O(n!) [where n is the number of valid moves in the hand] which is incredibly intensive, since there are often upwards of 50 valid moves on the surface, which means it would require approximately 3×10^{64} computations!
    - The runtime would be O(n!) because first the program loops n times, since it needs to test each of the n moves. Then, each of those n moves have approximately n-1 moves once that move had been taken from the hand - as we know, this pattern n * (n-1) * (n-2) * ... * 1 is n!
- I do have an idea of potentially using this, but only for *very small* hands with very few potential moves - this could be a potential extension I could build upon and somehow try to decrease the runtime complexity of

User Input Manager: `user_input.py`

Manages all the user input, by providing functions that take input from the user, and validate it using different means, including checking if the input is an integer, if it is within a numerical range, or if it is one of the acceptable values

- I used **pattern recognition** in my program to recognize it would be very useful to make functions specifically to deal with input and data validation, since I use it so regularly in the program to get user input
- These all use loops, which return (and thus break) if the input is valid, and continue looping otherwise

User Interface Manager: `user_interface.py`

`print_title()`

Prints the top of a box with a title for the user interface

- This is one example of many similar functions in this file, however it provides a good picture of the main idea for all functions (I had to use **decomposition**, **abstraction**, and **pattern recognition** to realize this, and make the code more concise since most of the functions have similar uses)
- This function accepts two arguments, `text` (a string containing the title), and `length` (default as `50`, an integer which indicates how wide the box should be in characters)

- Each line of the function adds a substring to the final outputted box title string: (1) upper-left corner of box, (2) space, (3) the actual title, (4) space, (5) an appropriate number of horizontal lines to fill the box, (6) upper-right corner of box
    - To determine (5), I used string manipulation, by multiplying `length` minus the length of the string after steps 1 through 4 minus 1 BY horizontal line
    - This allows the entire box to have width equal to the `length` variable specified

Extensions

- I could create a more advanced user interface/GUI for players to access, like going beyond just the terminal and actually creating an easy-to-access user interface (with buttons and such)