COMPUTATION II PROJECT

POKER CARD GAME

CARLOS AMORIM, 20211548
PROFESSOR AUGUSTO SANTOS
PROFESSOR DIOGO RASTEIRO
NOVA IMS 2025

CODE STRUCTURE AND ORGANIZATION

From the beginning, the project was structured in a modular way, breaking it down into separate files to make development and debugging easier and more manageable. Each module ('**.py**' file) has a specific responsibility, allowing the program to work on individual parts without creating dependency errors.

The modules are as follows:

- '**main.py**': is the main entry point for the game loop, user input, and the round logic;

- '**card.py**': defines the **Card** class (which represents a single playing card with a suit and a rank) with attributes for suit, value, and display formatting
    - **__init__(self, suit, rank)**: initializes card with suit and rank
    - **__str__(self)**: returns a string like 'ace of spades' for easy display
    - **comparison methods (__lt__, etc.)** to help sorting by rank and suit

  and the **Deck** class (which represents a full deck of 52 cards, handles the shuffling and drawing)
    - **__init__(self)**: creates a full ordered deck of cards
    - **shuffle(self)**: randomly shuffles the deck
    - **draw(self, count=1)**: draws count cards from the deck and removes them
    - **reshuffle(self)**: resets the deck and shuffles it again

- '**hand.py**': where the drawn cards are managed, sorted, and analyzed for poker hands. Defines the **Hand** class (which represents a player's hand (the cards dealt) and poker hand evaluation).
    - **__init__(self, cards)**: initializes the hand with a list of cards
    - **sort(self, algorithm)**: sorts the hand with a chosen sorting algorithm (merge, heap, quicksort, binary insertion)
    - **evaluate(self)**: checks the hand for poker combinations like pairs, flush, straight, full house, etc.
    - **get_hand_ranking(self)**: returns the detected poker hand ranking(s) in a readable format

  and the **PokerHandDetector** class (which represents the logic to detect poker hands in a set of cards).
    - **detect(cards)**: returns a dictionary with poker hand types and how many times each appears
    - **get_value_counts(cards)**: counts how many times each card value

appears
    - **is_flush(cards)**: checks if all cards have the same suit
    - **is_straight(cards)**: checks if cards form a consecutive sequence
    - **is_royal_flush(cards)**: checks for 10-J-Q-K-A of same suit
    - **is_straight_flush(cards)**: checks if hand is both straight and flush

- '**sorters.py**': where the sorting algorithms are implemented (Merge Sort via **def merge_sort(cards)**, Quick Sort via **def quick_sort(cards)**, Binary Insertion Sort via **def binary_insertion_sort(cards)**, and Heap Sort via **def heap_sort(cards)**). They are implemented from scratch, without using pythons built-in sorting functions. Defines the **SortManager** Class (which selects and runs the chosen sorting algorithm on a hand of cards)
    - **sort(cards, algorithm_name)**: applies the correct sort (merge, heap, binary, quick) based on user input

- '**ui.py**': manages user interaction in a clear and playful way and displays the cards using ASCII art via **def print_cards(cards)**;

- '**game.py**': controls how the game plays out from start to finish (draw -> sort -> detect -> repeat), keeps track of session statistics and handles reshuffling and replay/exit prompts. Defines the **Game** Class (which manages the overall game flow and user interaction)
    - **__init__(self)**: sets up deck, player hand, and game state
    - **start_round(self)**: deals a new hand and reshuffles deck if needed
    - **choose_sorting_algorithm(self)**: asks the user to select a sorting algorithm
    - **play_round(self)**: handles sorting the hand, evaluating, and showing results
    - **ask_replay(self)**: prompts the player if they want to play another round
    - **run(self)**: main loop controlling game rounds and replay until exit

- '**constants.py**': stores all the fixed values (the list of card suits, the list of card values as well as the numeric order of each card – for sorting purposes);
- '**__init__.py**': although not mandatory, it is part of best practices. This module improves formality and informs Python that the folder is a package
- '**tests.py**': this module is a testing sandbox and was used during development to check specific cases such as: deck draws and shuffles correctly (**test_deck**); hands are sorted properly (**test_hand_sorting**); special poker hands are detected accurately (**test_special_hands**) and it also includes specific hands to test edge cases like royal flushes, ace-low straights, full houses and invalid combinations

This separation made it easier to test specific logic (like sorting or hand detection) independently and reduced the amount of rewiring needed when gameplay flow changed.

KEY DESIGN AND IMPLEMENTATION DECISIONS

One of the first challenges encountered was how to represent cards in a way that would support both sorting and poker logic. At first using simple strings was considered (e.g., "J♠"), but parsing those during hand analysis or sorting would have been error prone. Instead, a Card class was defined with:

- value as an integer from 2 to 14 (with 2=2, 3=3, ..., A=14)
- suit as a string ('hearts - ♥' - , 'diamonds - ♦', etc.)

For the specific case where A=1 (ace-low straight) the implementation had to be done under the Poker Hand Detection logic in '**hand.py**' (refer to the method **def is_straight(cards)**).

This allows cards to be sorted to numerically and easily run comparisons without converting strings mid-game.

Storing and drawing from the deck also required some attention. A bug emerged during early testing: when drawing cards repeatedly without reshuffling the deck, index errors were encountered. To fix this, the deck had to be reshuffled at the beginning of every new round (as per the project requirements) and notify the user that the deck was being reshuffled. This both fixed the bug and made the game message clearer.

Integrating sorting also presented a problem: since Python passes lists by reference, sorting one hand was affecting other parts of the program during testing. This issue was solved by explicitly copying the hand before applying the sort, ensuring isolation.

Another notable decision was to separate the sorting algorithm selection from the sorting logic itself. Instead of embedding if-else chains inside the game loop, a mapping from user input was used to sorting functions. This made the code cleaner and also made it easier to add/remove algorithms later on.

Lastly, extra care was taken to preserve the polite tone of the game. The output was formatted in a graceful but playful language. All the relevant docstrings were also written.

SORTING ALGORITHM BEHAVIOR, DESIGN AND IMPLEMENTATION

The implementation of the sorting algorithms had to be done manually for Merge Sort, Quick Sort, Binary Insertion Sort, and Heap Sort. All were tested for correctness on several different card hands.

1. **Merge Sort**

Conceptually, Merge Sort is a divide-and-conquer algorithm that recursively splits the list into halfs, sorts them, and merges them back. It's stable and guarantees O(n log n) complexity. It was implemented with a '**merge_sort()**' function and a separate '**merge()**' helper. Challenges included managing list slices and avoiding side effects. Comparisons were based on '**card.value**', and not the objects themselves.

2. **Heap Sort**

Heap Sort relies on creating a max-heap and extracting the largest element iteratively. This implementation includes a '**heapify()**' helper to maintain the heap structure. It is in-place but not stable.

3. **Binary Insertion Sort**

This version of insertion sort uses binary search to find the correct insertion point. Though its time complexity is $O(n^2)$, it performs fewer comparisons. This algorithm was ideal for 5-card hands and gave the best practical performance. A custom binary search was created to find insertion indices and handled insertions by slicing, in order to avoid iteration issues.

4. **Quick Sort**

Quick Sort was the fourth sorting algorithm chosen to implement. It also uses the divide-and-conquer approach but splits the list around a pivot. The last element was used as pivot for simplicity. The time complexity is O(n log n) on average but $O(n^2)$ in the worst case. The implementation included a '**partition()**' helper. A major bug involved sorting equal card values improperly due to incorrect inequality operators.

During testing, all 4 sorting algorithms felt the natural and fast. Although it falls outside the scope of the project, for larger lists (e.g., 100+ cards), merge sort would likely be the best all-around option

POKER HAND DETECTION

Detecting poker hands was a core challenge in the project. The goal was to recognize **multiple hand types** in a single draw and provide **clear, accurate output**.

The logic works by:

- building a **frequency dictionary** to detect pairs, three-of-a-kinds, four-of-a-kinds, and full houses
- checking **sorted sequences** for straights
- checking **suit uniformity** for flushes and royal flushes
- combining both to identify **straight flushes**

Hands are checked **independently**, meaning multiple detections (e.g., a full house and two pairs) are possible. This was intentional to demonstrate the full detection logic, even if it's not standard scoring behavior.

The system uses a **counting-based strategy** and runs in **O(n)** time. Sorting the hand before analysis was crucial to ensure accurate straight detection. Frequency maps are built in a single pass, making the detection fast and efficient.

RUNNING THE PROJECT

The game can be run via any IDE or the terminal. Here's how to launch it:

1. Place all '**.py**' files in the same folder
2. Open a terminal or your IDE of choice and run: '**main.py**'
3. Follow the prompts to:
   - Enter the number of cards to be drawn
   - Choose a sorting algorithm
   - View the sorted hand and poker results
   - Replay or exit the program.
4. Read the program outputs carefully and have fun!

ASCII CARD DISPLAY

To improve the UI and make the game more engaging, the cards drawn by the user are displayed in ASCII-art format using the **print_cards()** function. This function takes a list of Card objects and visually prints them side-by-side in a way that resembles real playing cards. The suit symbols (♠, ♥, ♦, ♣) are used for visual realism, and spacing is adjusted so that values like "10" still align with single-character values like "A" or "K". This feature adds personality and a more casino-like UI to the game.

ANNEX

PROJECT REQUIREMENTS CHECKLIST

Suggested + necessary classes	**Y**
Implement sorting algorithms from scratch	**Y**
Time complexity analysis	**Y**
Game runs in python console	**Y**
Upon starting, create/instantiate a standard 52-card deck and shuffle it randomly, and inform user that a shuffling is being performed	**Y**
Allow the user to draw a hand of cards, specifying the desired number of cards (between 3 and 15)	**Y**
Display drawn hand	**Y**
User is prompted to choose a sorting algorithm	**Y**
Sort and display the drawn hand, along with time complexity (using big O notation)	**Y**
Drawn cards are sorted first by their values in ascending order	**Y**
After sorting, the game evaluates, detects and displays poker hands	**Y**
Update and display a counter for each type of hand encountered during the session	**Y**
Prompt the player to either: **(a) Play again: reshuffle the deck and repeat from step 2.** **(b) Exit: terminate the program and display final statistics.**	**Y**
Extra functionality: ASCII card design, improving UI and overall L&F	**Y**