Algorithms Project Report

Group 14

Chiara Rogani (20242141)
Constança Rocha (20241758)
Maiara Almada (20241723)
Tiago Silveira (20241698)

Data Science Bachelor
NOVA Information Management School
June 2025

1 Introduction

This project, developed for the Algorithms and Data Structures course, focuses on applying the core concepts learned in the classes, through the implementation of a card-based game.The goal is to simulate a standard 52-card deck and allow the user to draw a number of cards (hand), sort them, and check for the different combinations valid in a game of poker.

This program runs both in a Python console environment and in a Streamlit web graphical user interface (GUI). Allowing the user to select how many cards they want (between 3 and 15), pick a sorting method, receive the list of combinations found, and keep a tally of them.

2 Card, Deck, and Hand Implementation

2.1 Card Class

The `Card` class is responsible for representing individual playing cards. Each object has two primary attributes: value (such as '2', '10', 'J', 'Q', 'K', or 'A') and a suit (spades, clubs, hearts, or diamonds).

To facilitate the comparison between different cards and sorting them, to each card a numeric value was attributed, according to the following map:

Numeric Cards ('2' to '10') $\rightarrow$ the value is converted into an integer

Jack ('J') $\rightarrow$ 11

Queen ('Q') $\rightarrow$ 12

King ('K') $\rightarrow$ 13

Ace ('A') $\rightarrow$ 14

2.2 Deck Class

The `Deck` class is responsible for managing the full set of 52 unique playing cards used in the game. It generates and stores on a list the cards corresponding to every possible combination of value and suit.

The key functionalities of this class are shuffling and drawing a certain number of cards. The `shuffle` module uses the Ficher-Yates algorithm which consists in for each element in the `self.card` list generating a random integer (`randSeed`) between the values an index of the list might have. And, then, swapping it with the index `randSeed` element. This algorithm was chosen as it is well-known for guaranteeing that each permutation is equally likely (Zetzsche et al., 2025).

The `drawing` module selects the first `card_number` of cards in the deck, removes them from the deck, and returns the selected cards.

2.3 Hand Class

The `Hand` class deals with all things related to the game, per se, this class starts by generating a full deck of cards, shuffling it and drawing `card_number` cards (using the `Deck` class and its modules).

This class also aggregates all the sorting algorithms and the poker detection modules.

3 Sorting Algorithms

In this project, we implemented four sorting algorithms: **Heap Sort, Merge Sort, Binary Insertion Sort, and Quick Sort**. The user must choose which sorting algorithm before their hand is evaluated. Sorting the cards improves the efficiency of poker hand detection, especially for sequences like straights and flushes.

3.1 Heap Sort

Heap Sort is one of the implemented options, selected by choosing option 1. It works by building a max-heap from the card list and repeatedly moving the highest element to the end of said list and, in that way, sorting the list. It has an average time complexity of O(n log n) (Ali et al., 2021).

3.2 Binary Insertion Sort

Binary Insertion Sort, corresponding to option 2, uses binary search to find the ordered position for each card. This algorithms average time complexity is O(n log n) (Ahmad et al., 2013).

3.3 Merge Sort

Merge Sort, chosen via option 3, works by recursively dividing the hand into smaller sub lists, sorting them and merging the results. Just like the previous algorithms, its average time complexity is O(n log n) (Al-Kharabsheh et al., 2013).

3.4 Quick Sort

Quick Sort, implemented as option 4, selects a pivot element and partitions the list around it. It recursively applies the same process to the sub lists. Its average time complexity is O(n log n) (Chauhan and Duggal, 2020).

Table 1: Average time complexity of each sorting algorithm

Heap Sort	**Merge Sort**	**Binary Insertion Sort**	**Quick Sort**
O(n log n)	O(n log n)	O(n log n)	O(n log n)

4 Poker Detection Strategy

The core function `poker detection(self)` is responsible for identifying all standard poker hands from a set of cards `(self.cards)`. The function leverages a series of modular helper functions: `find_[...]`, each dedicated to detecting a specific hand type. The overall poker strategy is based on applying specific rule conditions to the set of cards.

The process is:

- Call all helper functions to search for specific hand types.
- Tally the combinations found using a global dictionary `hand_tally`.
- Print a descriptive message for each detected hand.

Each poker hand is defined by unique structural rules. The algorithm detects more specific and rare hands (e.g., Royal Flush) first, followed by more general ones (e.g., Pair). This prioritization prevents overlap and double counting:

- Royal Flush and Straight Flush: Detected using `find_straightFlush` and `find_straight` filtering for specific value sequences.
- Four of a Kind / Full House / Three of a Kind / Pair / Two Pairs: Detected by grouping cards by rank.
- Flush: Detected by counting suit occurrences.
- Straight: Detected by scanning for consecutive values, considering Ace as both high and low.

4.1 Key Principles

The implementation adopts a few practical considerations to ensure both correctness and performance. First, the cards are assumed to be pre-sorted, in order to optimize the detection of sequences (like straights or straight flushes) and makes the matching of identical ranks more efficient. To handle duplication, such as multiple cards of the same value, the algorithm optionally removes cards used in a valid hand from further evaluation. This prevents reuse within the same detection round and allows for accurate identification and tallying of multiple valid combinations in large card sets. Finally, the computational cost is kept low by most of the detection functions, which operate in a single pass or limited nested iterations to minimize computational cost.

4.2 Time Complexity Analysis

Table 2: Time complexity of each function

Function	Complexity	Explanation
`find_pair`	$O(n)$	While loop runs n times, in worst case
`find_two pairs`	$O(n)$	Runs `find_pair` plus `len(find_pair)`
`find_threeKind`	$O(n)$	While loop runs n times, in worst case
`find_straight`	$O(n^2)$	Nested loops - inner runs n times, outer runs $n+1$ times
`find_flush`	$O(n)$	Nested loops - outer runs 4 times, inner runs n times
`find_fullHouse`	$O(n^2)$	For each three of a kind, found it runs $2n$ times
`find_fourKind`	$O(n)$	While loop runs n times, in worst case
`find_straightFlush`	$O(n^2)$	It calls `find_straight` - n^2 and `find_flush` - n
`find_royalFlush`	$O(n^2)$	It calls `find_straightFlush` - n^2

4.3 Hand Tally System

To keep track of how many times each poker hand is detected during the round, the algorithm uses a `hand_tally` dictionary. This structure maintains a count for each possible hand type. Each time a hand is identified, its corresponding counter is incremented. This provides an overview of the combinations found in the poker hands across the card set.

5 Graphic User Interface Implementation

The Poker Hand Analyzer interface was developed using Streamlit, a Python library designed for building interactive web applications.

5.1 Interface Layout and Controls

The layout consists of a centered main display complemented by a functional sidebar, which allows users to configure the simulation parameters before analyzing a hand.

The sidebar provides intuitive input controls. Users can select the number of cards to draw using a slider(`st.sidebar.slider`) ranging from 3 to 15. Additionally, a dropdown menu (`st.sidebar.selectbox`) offers the choice of four different sorting algorithms, all defined in the backend `project.py` file. These include Heap Sort, Binary Sort, Merge Sort, and Quick Sort. Once the user configures these settings, a button initiates the entire process: drawing a new hand, applying the selected sort, and evaluating poker combinations.

5.2 Display Of The Cards

Cards are visually styled using HTML and Unicode symbols via `st.markdown()`, allowing red and black suits, card value display, and a clean UI with shadows and spacing. This enhances visual clarity and user engagement.

5.3 Deck Shuffling and Sorting

Upon clicking the "Deal Cards" button, an animated shuffle effect is triggered. A spinner and progress bar simulate the randomness and suspense of a real shuffle. Internally, this process involves creating a fresh deck object using `project.Deck()`, shuffling it, and drawing a set number of cards using the Hand class. The hand is then sorted according to the method chosen by the user, each of which corresponds to a dedicated function within the Hand class (e.g., hand.`heapSort()`] or `hand.quickSort()`)

5.4 Poker Detection Logic

After sorting, the program proceeds to evaluate the hand for poker combinations. Due to the original `pokerDetection()` method in project.py not returning values compatible with the interface, the detection logic was manually redefined within the `app.py` file. This detection system is capable of identifying all standard poker hands, including Royal Flush, Straight Flush, Four of a Kind, Full House, Flush, Straight, Three of a Kind, Two Pairs, and One Pair. Each recognized combination is instantly displayed on-screen using the `st.success()` component. For example, if a hand includes two tens and two fours, the app displays: Two pairs of 10 and 4. The output is not limited to just naming the combination. It also provides the card values involved and, when relevant (as in Royal Flushes, Straight Flushes, and Flushes), the suit and card range. This enhances the educational value of the tool by explaining not just what was found, but how.

5.5 Combination Tally

A final tally section uses `st.success()` or `st.info()` as a counter for each combination, ensuring that feedback is always clear.

6 Conclusion

This project provided a comprehensive approach to simulating and analyzing poker hands while integrating key computer science concepts such as sorting algorithms and data structures.

At the core of the application is a system capable of generating a random deck, drawing a hand of customizable size, applying various sorting algorithms, and detecting all significant poker hand combinations. These algorithms: Heap Sort, Merge Sort, Binary Sort, and Quick Sort; were not only implemented successfully but tested for correctness and performance in a real-use context.

From sorting performance to visual presentation, this project brought together both theory and practice in a clear and useful way. By using different sorting algorithms, each with its own logic and time complexity, the project made it easy to compare how they work. This connection between computer science concepts and a real card game made the learning experience more engaging and helped explain how sorting algorithms can be used in practical situations.

References

Ahmad, M., Ikram, A. A., Wahid, I., & Salam, A. (2013). Efficient sort using modified binary search-a new way to sort. *World Appl Sci J*, *28*(10), 1375–1378.

Ali, H., Nawaz, H., & Maitlo, A. (2021). Performance analysis of heap sort and insertion sort algorithm. *International Journal*, *9*(5).

Al-Kharabsheh, K. S., AlTurani, I. M., AlTurani, A. M. I., & Zanoon, N. I. (2013). Review on sorting algorithms a comparative study. *International Journal of Computer Science and Security (IJCSS)*, *7*(3), 120–126.

Chauhan, Y., & Duggal, A. (2020). Different sorting algorithms comparison based upon the time complexity. *International Journal Of Research And Analytical Reviews*, (3), 114–121.

Zetzsche, S., Tristan, J.-B., Lepoint, T., & Mayer, M. (2025). Verifying the fisher-yates shuffle algorithm in dafny. *arXiv preprint arXiv:2501.06084*.