Information and Analysis

Design

Defines and Structs

All struct fields are defined in order of increasing size. A card is represented by a “struct card” which holds a face value and a suit value. A player is represented by a “struct player” which holds the player’s name, their total score, and the values of the last card that they drew. The values for each suit and each nonnumerical face value, including penalty, have been predefined to maximize code readability.

Analysis

Theoretical Runtime Analysis

Dynamic Memory: getName()

The user can input as many characters as they want for a player’s name. The max size of the char array is predefined in miceli_cards.h to be 25 bytes. The algorithm is as follows:

- Let n be the characters of a player’s name
- For each n
    - if n is not NULL
        - if the current index of name[] is the end of the allocated block, then realloc()
    - push n into name[]
- return name[]

The total runtime of this algorithm is nondeterministic because it will depend on the memory management of the machine as n grows infinitely large. This is problematic for the worst case of infinitely large n, however, handing this issue is beyond the scope of this project.

This is the most potentially costly use of dynamic memory in this program, other uses like that of the deck do not have variable input sizes to analyze.

User Input

An uncontrollable source of time is waiting for user input. Like in waitForInput(), the program remains in a while loop until the ‘ENTER’ key is pressed. This is nondeterministic because it is entirely dependent on the user.

Card Game

We will examine the function play4 because we are interested in the worst case. The function call in main() for newPlayer() is constant time because it is called a deterministic number of times and contains no loops. Similarly, initDeck() is constant because it is only called once. The

only remaining function in main() besides play() that we need to worry about was already discussed: see Dynamic Memory.
Within the play functions, there are a few function calls which are constant so to avoid redundancy they will not be expanded in the following algorithm for play4.

- Let n be the number of rounds.
- Shuffle the deck
    - malloc a card pointer ptr
    - get the time with srand
    - for each 56 cards
        - set ptr to the top card in the deck
        - pull out a random card using rand() to get the index
        - stick that random card on top of the deck
        - place the old top card in the random spot
- For each player
    - takeTurn
    - go to the next card
- checkWin of player1 and player2
- checkWin of player3 and player4
- With the winner from 1v2 and the winner from 3v4, checkWin
- displayWin with the winning card
- addWin to the winner
- Print the scoreboard
- If player 1, 2, 3, or 4 have scores greater than or equal to 21
    - compare player1 and player 2
    - compare player 3 and player 4
    - compare the winner from 1v2 to the winner from 3v4
        - if we have a winner print congratulations
        - otherwise play4
- otherwise play4

shuffleDeck() gives us time complexity of 56*n + constant.
A player's turn gives us time complexity of 4*n + constant.
So the total time complexity of play4 is linear => O(60*n + constant)
The time complexity of the entire program is going to be O(n + getName()).
User Input has been left out of the total because average case is most likely constant. In the worst case the user will never press 'ENTER' so the time complexity would be (∞+).

Average Runtime

User Input

To account for user input, sleep(1) is used in place of asking users to input the number of players and player names. For a 4 player game it is a total of 5 seconds.

Play4

Using the testing files card_grind.c and card_grind_test.c, the average runtime was tested on the worst case, 4 players. For 10,000 runs, there is an average of 222 clock cycles per run. The 5 seconds for user input is not factored into the average clock cycle time, so this average is only with respect to the actions performed by the computer, therefore we can directly compare the average and the worst case.

Errors

Display

When running this program on OS X with LLVM interpretation of GCC, the displayed card prints as intended: a box of dashed lines with the face and suit value inside the box. However, an online REPL using Clang v7 displays displaced dashed lines, and therefore does not have the image of a card. The card values are correct but the aesthetic value is potentially lost on different platforms.

There is a known issue when running this program on Windows. The newline character, '\n', is frequently used in this program, but in Windows the equivalent character is '\r'. Modifications to the function getNumPlayers has fixed this issue's interference with the operation of the program. However, it is possible that the use of '\n' will cause discrepancy in the display between Windows and Linux/OSX

User Input

The most likely cause of error in this program is from user input. There are test cases included to catch these errors, print a message, and exit gracefully.