CPS506 Project Description

Suddenly, there was War!

Preamble

In this project, you will simulate a game of War. War is a simple card game between two players. A description of the game and its rules can be found here:

https://bicyclecards.com/how-to-play/war

Given the above description, there are still some situations that are ambiguous. They will be clarified in the game description below.

You will complete this project in three of the four languages we study in this course. The languages chosen are up to you. In addition to the general requirements of the project, each language comes with its own language-specific constraints. These specify the format of the input and output, as well as submission instructions for each language.

Aside from these requirements, anything you do inside your program is up to you. Use as many helper functions or methods as you want, use any syntax you find useful whether we covered it in class or not. There is one exception to this: you may not use any functionality that is not part of the base installation of each language. No 3rd party libraries. Your submission will be tested using out-of-the-box installations of each language.

Game description

The game starts with a shuffled deck of cards. The deck will be passed into your program already shuffled (details below). Your program will deal the cards in an alternating fashion to each player, so that each player has 26 cards. Each card is dealt on top of the previous, so that the top card of each pile after dealing will be the last card that was dealt.

In each round, both players reveal the top card of their pile. The player with the higher card (by rank) wins both cards, placing them at the bottom of their pile. Aces are considered high, meaning the card ranks in ascending order are 2-10, Jack, Queen, King, Ace.

If the revealed cards are tied, there is war! Each player turns up one card face down followed by one card face up. The player with the higher face-up card takes both piles (six cards – the two original cards that were tied, plus the four cards from the war). If the turned-up cards are again the same rank, each player places another card face down and turns another card face up. The player with the higher card takes all 10 cards, and so on.

When one player cannot play or draw a card because their pile is empty, they are the loser, and the other the winner. This applies during a war as well.

Technical details

Input: The input to your program, representing a shuffled deck of cards, will be a permutation of 52 integers, where each integer between 1-13 occurs four times. The integers in this permutation correspond to cards according to the following table (four kings, four tens, four threes, and so on). Notice that we don't bother representing the suit because the game of War doesn't require it.

1	2	3	4	5	6	7	8	9	10	11	12	13
Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King

The game: Your program will deal two piles from the input permutation. How you represent your piles is completely up to you. Once the piles are dealt, "play" the game in your program until one player can no longer draw cards. Once again, how you manage your piles during the game is completely up to you. Keep going until one player runs out of cards.

When cards are added to the bottom of a player's pile, they should be added in decreasing order by rank. That is, first place the highest ranked card on the bottom, then place the next highest ranked card beneath that. This is true of wars as well. If a player wins six cards as a result of a war, those cards should be added to the bottom starting with the highest rank and ending with the smallest. Ace has the highest rank, Two has the lowest.

Output: Your program will return the pile of the winning player. This pile should contain all 52 integers from the original input permutation and be in the *correct order* according to how the game played out.

Testing & Evaluation

Your code will be evaluated using an automated tester written by me. Therefore, it is of ***utmost importance*** that your code compile properly, handle input properly, and return results in the indicated format for each language. Do not deviate from the requirements or your code will fail the tester outright. Your code must ***compile and run*** as a baseline. Half-finished code that doesn't compile or is riddled with syntax errors will not be accepted.

To help you achieve the baseline of "code that works", you are provided with mix/cabal/cargo projects with a handful of simple test cases for Elixir/Haskell/Rust. For Smalltalk, I will provide a simple script you can paste into your Playground. The simple tests are to help you get started, but when I evaluate your final submission, I will be using a more sophisticated tester with more tests and more interesting shufflings.

Your grade for each project submission will be based primarily on the fraction of the tests for which your program returns the correct result. There are minor marks for code style, documentation, etc. The full rubric is below.

Marking Rubric

2 marks - Code style

Your code is clean like this:

```
#include <stdio.h>
int main()
{
   printf("Hello world!\n");
}
```

Not messy like this:

```
  #include <stdio.h>
        int main()      {
printf("Hello, World!\n");;;;
        /* lol */       }
```

3 marks - Documentation

Document functions, control structures, and scope blocks. Do not document every line. Documentation should be meaningful. Don't just say "This function deals the cards".

5 marks – Code runs, does something

Your code compiles (without warnings!) and runs without crashing, entering an infinite loop, etc. Your deal function must accept the input correctly and return a proper permutation of the input. The simple tester provides code to test this for you.

5 marks – Code passes provided tests

The simple tester provided for each language comes with a set of test cases. Passing these tests is worth five marks, awarded proportionally to the number of tests passed.

25 marks – Code passes mass tests

The remaining 25 marks are awarded to your code for passing all the tests my automated tester can throw at it. These marks are awarded proportionally. If your code passes 80% of the tests, you'll get 20/25, and so on.

40 marks total

Language requirements

The requirements of each language are evident from the code provided, but in brief, the input/output requirements are as follows:

Smalltalk: Create a class named **`War`** that implements a class method called **`deal`**. This method should accept and return an Array of 52 integers.

Elixir/Haskell: Implement a function called **`deal`** that accepts and returns a list of integers

Rust: Implement a function called **`deal`** that accepts an immutable reference to an array of **u8**, and returns ownership of a new array of u8.

If your code adheres to these constraints, it should play nice with my tester. Additionally, these constraints are already provided via function skeleton in the code provided for each language. All you must do is fill in the function, and not modify the parameters.

Submission

Projects must be submitted ***individually***.

Smalltalk submission: To submit your Smalltalk assignment, you will submit the entire Pharo image directory as an archive (zip/rar/7z/whatever). To find your Pharo image directory, select the image in the launcher that contains your assignment, and look at the bottom of the launcher screen to see the "Location" directory. Zip up ***everything*** in this folder – not just the image.

Elixir/Haskell/Rust submissions: Submit your war.ex, war.hs, or main.rs file on D2L. You do not need to submit the entire mix/cabal/cargo project.