C/CPS506 Assignment Description

Preamble

You will solve the same assignment using **two** of the four languages we study this semester. If you submit the assignment in more than two language, the best two will be chosen for your assignment marks.

In addition to the general requirements of the assignment, 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, 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.

General assignment description

For this assignment, you will write a program that deals two 2-card poker hands and a shared five card pool (the flop/turn/river for those who know these poker terms) according to the rules of Texas Holdem poker. Your program will determine the winning hand and return it. The rules of Texas Holdem and the various hand rankings can be found at the links below (ignore all talk of incremental betting/passing/folding, we aren't considering that):

https://www.partypoker.com/en/how-to-play/texas-holdem
https://www.fgbradleys.com/et_poker.asp

When determining the strength of each player's hand, you will consider the two cards that player was dealt, as well as the five cards present in the shared pool. The stronger hand will be returned as the winner.

Program Input

The input to your program will be the ***first 9 values*** in a permutation of the integers 1-52. This represents a ***shuffling*** of a standard deck of cards. The order of suits in an unshuffled deck are Clubs, Diamonds, Hearts, Spades. Within each suit, the ranks are ordered from Ace, 2-10, Jack, Queen, King. The table below shows all 52 cards and their corresponding integer values in a shuffling.

	1	2	3	4	5	6	7	8	9	10	11	12	13
Clubs	Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King
	14	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
Diamonds	Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King
	27	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**
Hearts	Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King
	40	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**
Spades	Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King

Thus, an input sequence that started with the integers [38, 48, 11, 6, …] would represent Queen of Hearts, 9 of Spades, Jack of Clubs, 6 of Clubs, and so on. Your program will never be tested with broken or invalid input. You do not have to error-check this.

Your program will accept this permutation as input and use it to deal two poker hands of two cards each in an alternating fashion. I.e., the first card goes to hand 1, the second card goes to hand 2, the third card goes to hand 1, fourth to hand 2. The remaining five cards will form the shared pool. Once dealt, your program will analyze each hand according to the rules from the websites above and decide a winner. When determining hand strength, each player considers their own cards as well as those in the shared pool. Effectively, each player has seven cards from which they will form the strongest possible hand.

Precise data types and format for the input and return values vary by language. See the "Language Requirements" section below for specifics.

Tie Breaking

According to the standard rules of poker, the ranks of the cards are used to decide the winner when both hands have the same strength. For example, if both hands are a flush, then the hand with the card of highest rank is declared the winner. If both hands have a pair, then the hand whose pair is higher wins. For example, a pair of Kings beats a pair of Sevens. If both hands have the same pair, i.e. each hand has a pair of threes, then the hand with the next highest card wins (called the *kicker*).

It is possible for two hands to remain tied even after all tie-breaking mechanisms based on rank are considered. An absurd example would be if the five cards in the shared pool formed a royal flush, in which case both players would have the exact same best hand (the royal flush). **You may rest easy in the knowledge that your program will not be tested on such inputs.** You may assume that all inputs will produce a clear and unambiguous winner. To put it in poker terms, you don't have to worry about any input that would result in the players splitting the pot.

Language Requirements

For all languages, your program will be driven at the top level by a function/method called **`deal`** that accepts the permutation array described above as input. This function will return the winning hand in the form described for each language below. Anything else you do inside this **`deal`** function is completely up to you. Except for the output format (which ***MUST*** be adhered to), which data types and structures you use internally to represent cards, hands, etc. is all completely up to you.

1) Smalltalk requirements:

You will create a class called **`Poker`**, with an instance method called **`deal:`** that accepts a simple integer array as input and returns the winning hand. Its usage will be as follows:

```
| deck winner shuf |
deck := Poker new. "Create new Poker object"
shuf := #(1 2 3 4 5 6 7 8 9) "Sample shuffling"
winner := deck deal: shuf. "Call deal: method with shuf as arg"
```


Your deal method will return the winning hand as an array of strings, where each element is a concatenation of the rank and the suit. The suit must be capitalized, and face cards are represented **numerically** (11=Jack, 12=Queen, 13=King). An Ace is represented numerically as 1. For example, if the winning hand was a straight flush from 3-7 in spades, it would be returned as the following array:


```
#('3S' '4S' '5S' '6S' '7S')
```


2) Elixir requirements:

In a single Elixir file called Poker.ex, define a module called **`Poker`**, with a function called **`deal`** that accepts a list of integers as an argument and returns the winning hand. Its usage will be as follows:

```
winner = Poker.deal [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The winning hand must be returned as a list of strings, where each element is a concatenation of the rank and the suit. The suit must be capitalized, and face cards are represented **numerically** (11=Jack, 12=Queen, 13=King). An Ace is represented numerically as 1. For example, if the winning hand was a straight flush from 3-7 in spades, it would be returned as the following array:

```
["3S", "4S", "5S", "6S", "7S"]
```

3) Haskell requirements

In a single Haskell file called Poker.hs, define a module called **Poker**, with a function called **deal** that accepts a list of integers as an argument and returns the winning hand. Its usage will be as follows:

```
winner = Poker.deal [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The winning hand must be returned as a list of strings, where each element is a concatenation of the rank and the suit. The suit must be capitalized, and face cards are represented **numerically** (11=Jack, 12=Queen, 13=King). An Ace is represented numerically as 1. For example, if the winning hand was a straight flush from 3-7 in spades, it would be returned as the following array:

```
["3S", "4S", "5S", "6S", "7S"]
```

Your Poker.hs file must load cleanly into GHCi. When your submission is tested, the deal functions will be called from a separate test function. Please ensure your Poker module, and any additional modules you create, compile correctly.

4) Rust requirements

In a single Rust file called Poker.rs, write a function called **deal** that accepts an array of integers as an argument and returns the winning hand. In Rust, the typing of the input argument and the return value are very important. Rust will ***NOT*** implicitly convert anything for you, so pay extra attention to this! The usage of the deal function must be as follows:


```
let perm:[u32;9] = [1, 2, 3, 4, 5, 6, 7, 8, 9];
let winner:Vec<String> = deal(perm);
```

Your deal function should accept an array of 9 unsigned integers, and your deal function should return a ***vector*** of five Strings. The strings should be formatted in the same way as all previous assignments.

Your submission should not include a **main()** function. The tester will implement **main()**. Finally, the deal function should be declared as public using the **pub** keyword. The required signature for your deal function is below:

```
pub fn deal(perm:[u32;9])->Vec<String>
{
      // Your code goes here
}
```

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 *work* as a baseline. Half-finished code that doesn't compile or is riddled with syntax errors will not be considered.

To help you achieve the baseline of "code that works", I will be releasing a simple tester for each language that will evaluate your program for a handful of test cases. The full tester will work the same way as this simple tester, except it will run a far greater number of tests.

Beyond that, your grade for each assignment submission will be based solely on the fraction of the tests for which your program returns the correct winning hand. There are no marks for code style, documentation, or anything of that sort.

Here are some examples of inputs and their expected outputs, using Elixir style lists. These are the same test cases found in the simple testers.


```
[ 9,  8,  7,  6,  5,  4,  3,  2,  1  ] -> [ "2C",  "3C",  "4C",  "5C",  "6C" ]
[ 40, 41, 42, 43, 48, 49, 50, 51, 52 ] -> [ "10S", "11S", "12S", "13S", "1S" ]
[ 40, 41, 27, 28, 1,  14, 15, 43, 29 ] -> [ "1C",  "1D",  "1H",  "1S" ]
[ 30, 13, 27, 44, 12, 17, 33, 41, 43 ] -> [ "4D",  "4H",  "4S" ]
[ 27, 45, 3,  48, 44, 43, 41, 33, 12 ] -> [ "2S",  "4S",  "5S",  "6S",  "9S" ]
[ 17, 31, 30, 51, 44, 43, 41, 33, 12 ] -> [ "4D",  "4H",  "4S" ]
[ 17, 39, 30, 52, 44, 25, 41, 51, 12 ] -> [ "12C", "12D", "12S", "13H", "13S" ]
[ 11, 25, 9,  39, 30, 48, 3,  49, 45 ] -> [ "10S", "11S", "12D", "13H", "9S" ]
[ 50, 26, 39, 3,  11, 27, 20, 48, 52 ] -> [ "11C", "11S", "13H", "13S" ]
[ 40, 52, 46, 11, 48, 27, 29, 32, 37 ] -> [ "1H",  "1S" ]
```


Submission

You may work in teams of **two (2)** for the assignment, with one caveat: you may not work with the same partner more than once. If two people work together for the Smalltalk assignment, those same two people may not work together for any of the other languages.

Smalltalk submission: To submit your Smalltalk assignment, you will submit the entire Pharo image directory as an archive (zip/rar/7z/whatever). Our TAs will give a demo on how to do this in a future lab, and you may ask them for help with the submission.

Elixir/Haskell/Rust submissions: Submit your Poker.ex, Poker.hs, or Poker.rs file on D2L. If you're working with a partner, only one partner submits. Clearly indicate, in both your submission and your Poker source file (as a comment), who the two team members are.