IE523: Financial Computing
Fall, 2016

Programming Assignment 4: How much would you pay to play this card game?

Due Date: 22 September 2016

©Prof. R.S. Sreenivas

This programming exercise is meant to get you thinking about recursive algorithms for pricing options and other financial instruments. We will use a card-game to motivate the algorithm development. The problem statement is as follows:

> I shuffle a deck of cards (with an equal number of **red** and **black** cards) and start dealing them face up.
>
> After any card you can say "**stop**", at which point I pay you $1 for every **red** card dealt and you pay me $1 for every **black** card dealt.
>
> What is your optimal strategy, and how much would you pay to play this game?

You are going to write a C++ program, which uses recursion, that takes as command-line input the number of cards in the deck, and comes up with the *no-arbitrage price*[1] for this game.

[1] *Arbitrage* occurs when you can make a profit that is in excess of the *risk-free rate of return*, which is zero for this problem.

You have an *arbitrage opportunity* if you can create a portfolio of zero value today, but this portfolio has a positive value in the future with a positive probability, and a negative value in the future with zero-probability.

The "*no-arbitrage principle*" suggests that there are no arbitrage opportunities in any market. This "*there is no free-lunch*" assumption is the corner-stone of classical finance theory (cf. section 3.2, [**?**]).

기대값 계속 계산.

Hints

1. The no-arbitrage price is at least zero. Because, you can wait till all cards are dealt (and there is the same number of **red** and **black** cards dealt; and you owe nothing).

2. You should ask for another card only if the expected value of continuing to play is greater than the amount of money already won by you. Otherwise, you should say "**stop**."

3. If value(#red cards left, #black cards left) is a function that computes the value of the game to you at any point, then

$$\begin{aligned} &\text{value}(\#\textbf{Red cards left}, \#\textbf{Black cards left})) = \\ &\max\{(\text{Prob of }\textbf{Red Card}\text{ drawn}) \times \text{value}(\#\textbf{Red cards left-1}, \#\textbf{Black cards left}))+ \\ &\quad (\text{Prob of }\textbf{Black Card}\text{ drawn}) \times \text{value}(\#\textbf{Red cards left}, \#\textbf{Black cards left - 1})), \\ &\qquad (\#\textbf{Black cards left - }\#\textbf{Red cards left}) \end{aligned} \tag{1}$$

   (??**Why**??)

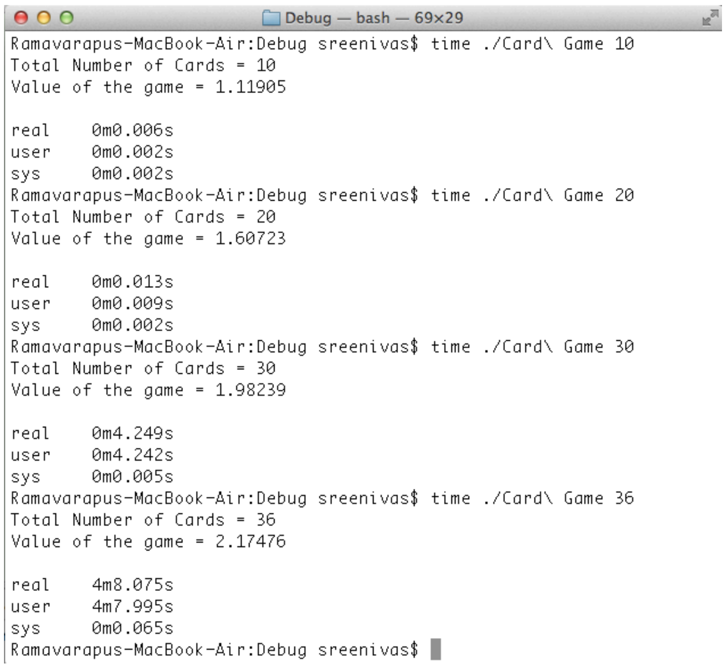
4. Think of a recursive implementation (after presenting an appropriate justification) for equation 1 shown above.

   (a) A naive implementation of the recursion will run into run-time problems (cf. figure 1) when the size of the deck exceeds 35.

   (b) Look up the method of memoization to come up with a faster implementation of the recursion, which can handle large card deck size with ease (cf. figure 2). Or, you could just pay attention when I cover this in class ◡̈.

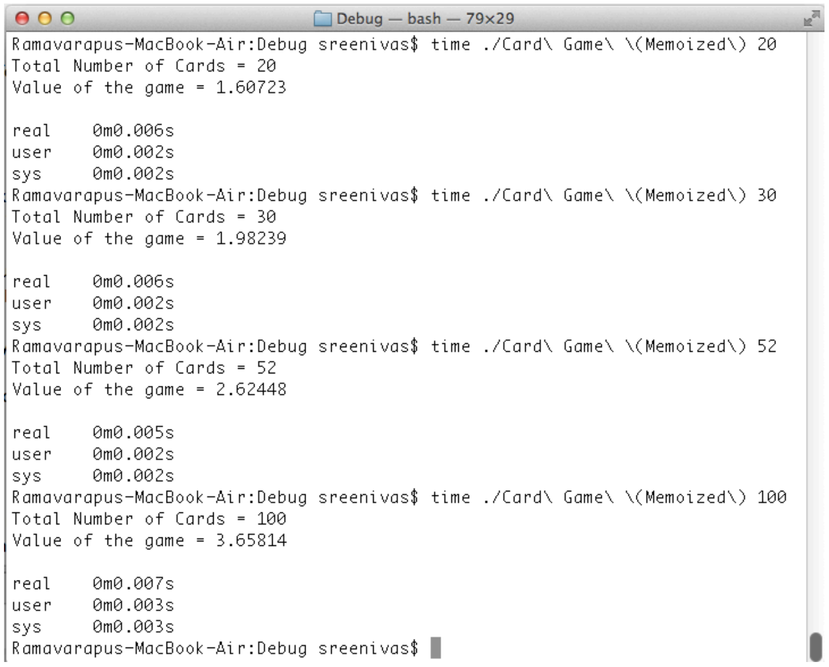
Here is what I want from you for this programming assignment

1. C++ code with appropriate Class definitions that takes the size of the deck as input and returns the arbitrage-free price for playing the game. Your implementation should be able to handle a deck with 52 cards, at least.



```
Ramavarapus-MacBook-Air:Debug sreenivas$ time ./Card\ Game 10
Total Number of Cards = 10
Value of the game = 1.11905

real    0m0.006s
user    0m0.002s
sys     0m0.002s
Ramavarapus-MacBook-Air:Debug sreenivas$ time ./Card\ Game 20
Total Number of Cards = 20
Value of the game = 1.60723

real    0m0.013s
user    0m0.009s
sys     0m0.002s
Ramavarapus-MacBook-Air:Debug sreenivas$ time ./Card\ Game 30
Total Number of Cards = 30
Value of the game = 1.98239

real    0m4.249s
user    0m4.242s
sys     0m0.005s
Ramavarapus-MacBook-Air:Debug sreenivas$ time ./Card\ Game 36
Total Number of Cards = 36
Value of the game = 2.17476

real    4m8.075s
user    4m7.995s
sys     0m0.065s
Ramavarapus-MacBook-Air:Debug sreenivas$
```

Figure 1: Sample output. Notice the growth in running-time as the #cards in the deck is increased. We have think of something to overcome this issue. We want to be able to handle decks of size 52 or higher! See sample output in figure 2.



```
Ramavarapus-MacBook-Air:Debug sreenivas$ time ./Card\ Game\ \(Memoized\) 20
Total Number of Cards = 20
Value of the game = 1.60723

real    0m0.006s
user    0m0.002s
sys     0m0.002s
Ramavarapus-MacBook-Air:Debug sreenivas$ time ./Card\ Game\ \(Memoized\) 30
Total Number of Cards = 30
Value of the game = 1.98239

real    0m0.006s
user    0m0.002s
sys     0m0.002s
Ramavarapus-MacBook-Air:Debug sreenivas$ time ./Card\ Game\ \(Memoized\) 52
Total Number of Cards = 52
Value of the game = 2.62448

real    0m0.005s
user    0m0.002s
sys     0m0.002s
Ramavarapus-MacBook-Air:Debug sreenivas$ time ./Card\ Game\ \(Memoized\) 100
Total Number of Cards = 100
Value of the game = 3.65814

real    0m0.007s
user    0m0.003s
sys     0m0.003s
Ramavarapus-MacBook-Air:Debug sreenivas$
```

Figure 2: Sample output of an implementation that can handle cases where the #cards in the deck is large. See sample output in figure 1 for a comparison of run-times.