Calculateur de probabilités pour le poker

Noms & Prénoms des auteurs du document :

DE JESUS Logan

GESTIN Loïc

RICHARD Mathis

Sujet : Le but de ce projet est d'implémenter un calculateur de probabilités pour le poker, et en particulier la variante Texas Hold'em (2 cartes fermées,5 cartes communes).

Structure des cartes :

Pour représenter les cartes nous avons choisi de créer 2 Enum, une pour la couleur de la carte et l'autre pour sa valeur, il sera, grâce à cette structure, très facile de créer une carte par la suite.

```
enum Couleur:
  case PIQUE
  case COEUR
  case CARREAU
  case TREFLE

enum Nombre:
  case AS
  case DEUX
  case TROIS
  case QUATRE
  case CINQ
  case SIX
  case SEPT
  case HUIT
  case NEUF
  case DIX
  case VALET
  case DAME
  case ROI
```

Ensuite, nous avons créé une classe PlayingCard qui regroupe les 2 enum et qui étend des comparateurs (qui serviront à comparer les cartes puis les mains de tous les joueurs). Nous avons aussi implémenté un toString pour l'affichage des cartes et surtout la fonction valueOf, qui va donner une valeur à chaque carte selon son nombre (As, Roi, 10, 5…) ce qui permettra de faire des opérations complexes dans les prochaines fonctions à implémenter.

```
class PlayingCard(val nombre: Nombre, val couleur: Couleur) extends Ordered[PlayingCard]{

    override def compare(that: PlayingCard): Int = ComparePC.compare(this,that);

    override def toString: String = stringValue + suitValue
      def stringValue: String = this.nombre.match {
        case Nombre.AS => "A"
        case Nombre.VALET => "V"
        case Nombre.DAME => "D"
        case Nombre.ROI => "R"
        case _ => (valueOf+1).toString
      }
      def suitValue: String = this.couleur match {
        case Couleur.TREFLE => "♣"
        case Couleur.PIQUE => "♠"
        case Couleur.COEUR => "♥"
        case Couleur.CARREAU => "♦"
      }
```

Nous regroupons ensuite toutes les différentes cartes du poker dans une liste de PlayingCard (FrenchDeck) et nous retirons aléatoirement des cartes de cette liste pour les donner aux différents joueurs, il est aussi possible de directement donner des cartes spécifiques aux joueurs pour faire des tests par exemple.

Les différentes mains :

Toutes les fonctions déterminant si pour une main de n carte(s), il y a un brelan, une suite, ou n'importe quelle main possible dans le poker; ont été implémentées.

Nous nous servons de 3 listes pour qui vont rentrer dans ces fonctions, une liste qui donne les occurrences de nombre (As, Roi…) dans la main du joueur, une autre qui donne les occurrences de couleurs (Cœur, Pique…) et finalement la liste avec les cartes du joueur (PlayingCard) triée de la carte la plus haute à la plus basse.

```
def ListeOccurencesNombre: List[Int] = Nombre.values.toList.map(numero => this.cartes.count(_.nombre == numero));
def ListeOccurencesCouleur: List[Int] = Couleur.values.toList.map(coul => this.cartes.count(_.couleur == coul));

def ListePlayingCardSorted: List[PlayingCard]= this.orderedcartes;
```

Nous utilisons ces 3 listes selon la fonction, par exemple, pour la fonction qui détermine si la main possède une couleur, nous n'utilisons que la liste qui donne les occurrences de couleur :

```
/**Couleur*/
def isFlush: Boolean = isFlushAux(ListeOccurencesCouleur)
def isFlushAux(l : List[Int]): Boolean = l match
  case Nil => false
  case _ => if(l.head==5) true else isFlushAux(l.tail)
```

Et pour connaître si une main a une paire nous utilisons seulement la liste qui contient les occurrences des cartes :

```
/**Paire*/
def isPair: Boolean = isPairAux(ListeOccurencesNombre, count = 0)
def isPairAux(l : List[Int],count: Int): Boolean = l match
  case Nil => count==1
  case _ => if(l.head==2) isPairAux(l.tail,count+1) else isPairAux(l.tail,count)
```

De même pour la liste qui contient directement les cartes, nous l'utilisons pour les fonctions les plus complexes comme la suite de couleur ou encore le fullHouse.

Comparateur des mains :

Pour comparer 2 mains différentes nous utilisons des comparateurs et une méthode “valueOf” qui va donner une valeur au type de mains que nous obtenons. Par exemple, si une main est une suite, elle aura la valeur 5, et si une autre main est une paire, elle aura la valeur 2. Grâce à cette méthode nous pouvons comparer rapidement 2 mains. Si 2 mains ont la même valeur, alors nous déterminons s’il faut les comparer avec la carte haute ou s'il y a égalité.

```
def valueOf: Int =
  if isRoyalFlush then 10
  else if isStraightFlush then 9
  else if isFourOAK then 8
  else if isFullHouse then 7
  else if isFlush then 6
  else if isStraight then 5
  else if isThreeOAK then 4
  else if isTwoPair then 3
  else if isPair then 2
  else 1
```

Elle est très facilement compréhensible, et très pratique pour déterminer ce qu’il faut quand deux mains ont la même valeur (2 suites, 2 paires…) et qu’il faut vérifier sa carte haute. Il reste néanmoins certains problèmes, si on obtient 2 Royal Flush, il va quand même essayer de les comparer par carte haute, ce qui n’est pas nécessaire.

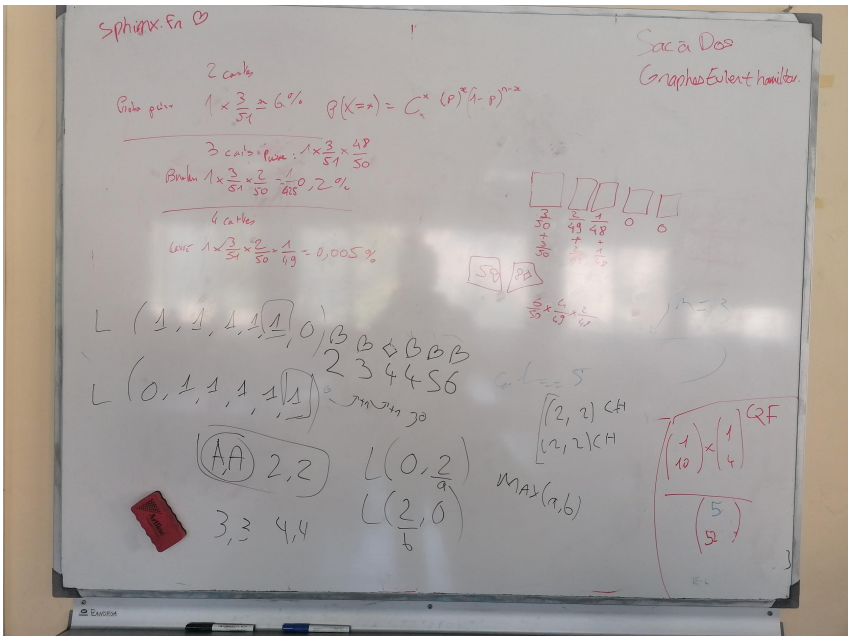
```
object ComparePC extends scala.math.Ordering[PlayingCard] {
  def compare(a: PlayingCard, b: PlayingCard): Int =
    if (a.valueOf > b.valueOf) 1
    else if (a.valueOf < b.valueOf) -1
    else 0
}

object ComparePH extends scala.math.Ordering[PokerHand]{
  def compare(a: PokerHand, b: PokerHand): Int =
    if(a.valueOf > b.valueOf) 1
    else if (a.valueOf < b.valueOf) -1
    else {
      val d: Int = compareAux(a.orderedcartes.head.valueOf,b.orderedcartes.head.valueOf)
      d
    }
```

Calculateur des probabilités :

Pour calculer la fonction qui calcule les probabilités :

“f*onction qui prend un ensemble de n cartes distribuées pour un joueur (ouvertes ou fermées), un nombre m de cartes restant à distribuer, et calcule, pour chaque main, la probabilité que le joueur obtienne cette main à la fin de la distribution (sim= 0, alors il s’agit simplement de déterminer quelle main le joueur a, sinon il faudra prendre en compte les différentes possibilités en fonction des cartes restant dans le jeu) ”*

Nous avons premièrement pensé à vérifier toutes les possibilités de distribution de carte et d’évaluer le type de main (brelan, paire …), mais un problème est apparu assez vite, si nous utilisons dans cette fonction n = 0 et m = 12, il faut vérifier un total de 12 parmi 52 possibilités, c'est-à-dire 206379406870 mains différentes ! La puissance de calcul étant insuffisante pour n’importe quel ordinateur (si on prend m = 20 cela augmente exponentiellement) Nous avons essayé de réfléchir à d’autres possibilités de programmer cette fonction.



Et après avoir réfléchi à des techniques de dénombrement et de factorielles, nous n’avons pas trouvé de solution meilleure que la première.

Il fallait donc concevoir un programme qui permet d’évaluer toutes les mains possibles mais nous n’avons pas réussi à créer une fonction qui ne demande pas trop de ressources à l’ordinateur, il y a beaucoup trop de possibilités à évaluer.

Pour les 2 fonctions suivantes à programmer, il aurait suffit d’utiliser la fonction qui calcule la probabilité d’avoir chaque main (avec n et m) et de simplement les comparer entre les joueurs et cela aurait été tout ce qui nous reste à faire.

Idem pour la dernière fonction : "fonction *qui prend les deux cartes fermées de chaque joueur, le tableau(0, 3, 4 ou 5 cartes), et calcule pour chaque joueur la probabilité d'avoir la meilleure main à la fin de la distribution."*

Il aurait fallu faire un appel de la fonction qui compare la probabilité de chaque joueur (avec n = 0, 3,4 ou 5 et m = 5,2,1 ou 0 pour la fonction qui calcule la probabilité) et de comparer et donnée le résultat de chaque main selon l'étape (0,3,4 ou 5 cartes).

Conclusion :

Même si nous n'avons pas réussi à implémenter toutes les fonctions, cela restait un projet intéressant, tant sur l'application du Scala, que sur les probabilités du poker en elle-même. C'est un défi qui aurait mérité plus de temps et d'approfondissement de notre part, mais nous en sommes sortis bien plus à l'aise vis-à-vis du langage Scala, de ses problématiques et des spécificités de sa syntaxe, ainsi que celles du lamda-calcul.

Nous pensons utiliser la programmation fonctionnelle dans nos futurs programmes, certains de nos membres utilisent dores et déjà cet outil dans le Java avec l'API Stream qui implémente la logique de la programmation fonctionnelle et des lambdas.