# Exercices sur les structures et les tableaux : Probabilité des mains au Poker Texas Hold'em

Dans ce TD/TP, on cherche à vérifier expérimentalement que les probabilités des mains au Poker Texas Hold'em (meilleure main de 5 cartes choisies parmi 7) sont celles écrites dans Wikipédia :

| Hand | % |
|---|---|
| Straight flush | 0.0311 |
| Quads (4 of a kind) | 0.168 |
| Boat (Full house) | 2.60 |

| Hand | % |
|---|---|
| Flush | 3.03 |
| Straight | 4.62 |
| Trips (3 of a kind) | 4.83 |

| Hand | % |
|---|---|
| Two pairs | 23.5 |
| Pair | 43.8 |
| High Card | 17.4 |

Pour cela, on effectue des centaines de milliers de fois la même opération : on mélange un jeu de 52 cartes, on en extrait les 7 dernières, et on calcule la meilleure main de 5 cartes parmi ces 7. On compte ainsi tous les types de mains obtenus, et on vérifie que l'on est bien proche des probabilités annoncées :

```
$ ./poker
...
Ks Kh Kc 8h 8d (Boat)
   Hi-Card: 156762 / 900000 = 17.4180% (error: +0.0061)
      Pair: 394679 / 900000 = 43.8532% (error: +0.0307)
 Two Pairs: 211434 / 900000 = 23.4927% (error: -0.0029)
     Trips:  43245 / 900000 =  4.8050% (error: -0.0249)
  Straight:  41845 / 900000 =  4.6494% (error: +0.0301)
     Flush:  27093 / 900000 =  3.0103% (error: -0.0152)
      Boat:  23171 / 900000 =  2.5746% (error: -0.0215)
     Quads:   1521 / 900000 =  0.1690% (error: +0.0009)
 Str-Flush:    250 / 900000 =  0.0278% (error: -0.0033)
```

En dehors du module `Main` contenant la fonction `main()`, le programme a 3 modules principaux :

- `Card` pour la gestion d'une carte par une structure du même nom :

```c
#define RANK_COUNT 13
#define SUIT_COUNT  4
char const RANK_PIPS[RANK_COUNT+1]= "23456789TJQKA"; // from Deuce to Ace.
char const SUIT_PIPS[SUIT_COUNT+1]= "cdhs"; // clubs, diams, hearts, spades.

typedef struct Card {
  int rank; // in range 0 to 13-1
  int suit; // in range 0 to  4-1
} Card;
```

- `Deck` pour la gestion d'un paquet de cartes par une structure du même nom :

```c
#define DECK_CAPACITY (RANK_COUNT * SUIT_COUNT)

typedef struct Deck {
  Card cards [DECK_CAPACITY];
  int length;
} Deck;
```

- `Hand` pour l'analyse d'une main avec une énumération du même nom :

```c
typedef enum Hand {
  HAND_UNKNOWN= -1,
  HAND_HICARD, HAND_PAIR,     HAND_2PAIRS,
  HAND_TRIPS,  HAND_STRAIGHT, HAND_FLUSH,
  HAND_BOAT,   HAND_QUADS,    HAND_STRFLUSH,
  HAND_COUNT
} Hand;

char const * HAND_NAMES[HAND_COUNT]= {
  "Hi-Card", "Pair",     "Two Pairs",
  "Trips",   "Straight", "Flush",
  "Boat",    "Quads",    "Str-Flush"
};
```

Trois autres modules `CardTest`, `DeckTest` et `HandTest` testent les fonctions des modules respectifs.

# 1 Le module `Card`

Le module `Card` fournit les fonctions suivantes :

Les prédicats `Card_rankIsValid()` et `Card_suitIsValid()` testent respectivement si `rank` et `suit` sont un rang valide et une couleur valide (contrôle de plage numérique). Elles sont utilisées comme assertion par la fonction `Card_make()` qui fabrique une carte d'un rang et d'une couleur donnée.

```
bool Card_rankIsValid (int rank);
bool Card_suitIsValid (int suit);
Card Card_make (int rank, int suit);
```

Les prédicats `Card_isPaired()` et `Card_isSuited()` testent si deux cartes ont respectivement appariées (de même rang) ou assorties (de même couleur). Le prédicat `Card_equals()` teste l'égalité de deux cartes (ce qui n'arrive normalement jamais lorsqu'on joue avec un seul paquet).

```
bool Card_isPaired (Card card1, Card card2);
bool Card_isSuited (Card card1, Card card2);
bool Card_equals   (Card card1, Card card2);
```

Le fonctions de conversion `Card_rankFromPip()` et `Card_suitFromPip()` retournent respectivement le rang et la couleur d'un symbole. Par exemple `'K'` est le symbole du Roi (*King*) qui est de rang 11, et `'c'` est le symbole du trèfle (*clubs*) qui est la couleur 0. La valeur bidon `-1` est retournée lorsque le symbole est inconnu. Ces fonctions sont utilisées par `Card_makeFromPip()`, qui fabrique et retourne une carte à partir de son symbole `pip` à deux caractères. Par exemple, si `pip` vaut `"Kc"`, la fonction fabrique et retourne le Roi de trèfle (*King of clubs*).

```
int  Card_rankFromPip (char rankPip);
int  Card_suitFromPip (char suitPip);
Card Card_makeFromPip (char const pip[]);
```

Les fonctions `Card_rankPip()` et `Card_suitPip()` retournent respectivement le symbole du rang et de la couleur d'une carte. Par exemple, elle retournent respectivement `'K'` et `'c'` pour le Roi de trèfle (*King of clubs*). Le prédicat `Card_pipEquals()` teste si une carte est celle correspondant à un symbole donné. Par exemple, elle renvoie `true` pour le Roi de trèfle et le symbole `"Kc"`. La fonction d'affichage `Card_printPip()` affiche le symbole d'une carte sur le flux de sortie spécifié. Par exemple, elle affiche la chaîne `"Kc"` pour le Roi de trèfle, sans retour à la ligne.

```
char Card_rankPip   (Card card);
char Card_suitPip   (Card card);
bool Card_pipEquals (Card card, char const pip[]);
void Card_printPip  (Card card, FILE * file);
```

Les fonctions de comparaison `Card_compareByRankOnly()` et `Card_compareBySuitOnly()` comparent respectivement les rangs et les couleurs de deux cartes. Le résultat est `< 0`, `= 0`, ou `> 0` selon l'ordre relatif des deux rangs ou des deux couleurs, comme d'usage pour les fonctions de comparaison. Noter que `Card_compareByRankOnly()` ignore les couleurs, et `Card_compareBySuitOnly()` ignore les rangs.

```
int Card_compareByRankOnly  (Card card1, Card card2);
int Card_compareBySuitOnly  (Card card1, Card card2);
```

A contrario, `Card_compareByRankFirst()` compare d'abord les rangs, puis les couleurs en cas d'égalité de rang, et `Card_compareBySuitFirst()` compare d'abord les couleurs, puis les rangs en cas d'égalité de couleur.

```
int Card_compareByRankFirst (Card card1, Card card2);
int Card_compareBySuitFirst (Card card1, Card card2);
```

La fonction de comparaison `Card_compareArrayByRankOnly()` compare deux tableaux de cartes de même longueur `n` pour l'ordre lexicographique induit par la comparaison des rangs. Cette fonction permet de comparer deux mains de même nature, par exemple deux *flushes* ou deux *straights*, lorsque les cartes des mains sont triées par rang décroissant.

```
int Card_compareArrayByRankOnly (Card const cards1[], Card const cards2[], int n);
```

Le prédicat `Card_isRegularConnector()` teste si `loCard` et `hiCard` sont connectées, c'est-à-dire de rangs contigus (`loCard` étant la plus petite). Le prédicat `Card_isRegularSuitedConnector()` teste si `loCard` et `hiCard` forment un connecteur assorti (de même couleur). Ces fonctions sont utilisées pour détecter un *straight* et *straight-flush* régulier. Ici, "*régulier*" signifie que les As jouent à leur rang régulier, et donc que As-Roi forment un connecteur régulier, mais pas As-Deux.

```
bool Card_isRegularConnector        (Card hiCard, Card loCard);
bool Card_isRegularSuitedConnector  (Card hiCard, Card loCard);
```

Le cas spécial du connecteur As-Deux est traité séparément. Les prédicats `Card_isAce()` et `Card_isDeuce()` testent qu'une carte est respectivement un As ou un Deux. Enfin, les prédicats `Card_isAceDeuceConnector()` et `Card_isAceDeuceSuitedConnector()` testent que deux cartes forment respectivment un connecteur As-Deux et connecteur As-Deux assorti (de meme couleur). Ces fonctions sont utilisées pour détecter un *wheel straight* ou *wheel straight-flush* : *5432A*.

```
bool Card_isAce   (Card card);
bool Card_isDeuce (Card card);
bool Card_isAceDeuceConnector       (Card hiCard, Card loCard);
bool Card_isAceDeuceSuitedConnector (Card hiCard, Card loCard);
```

## 2 Le module `Deck`

Le module `Deck` fournit les fonctions suivantes :

Les prédicats `Deck_isEmpty()` et `Deck_isFull()` testent respectivement qu'un paquet est vide (sans cartes) ou plein (capacité de stockage atteinte). Ces fonctions servent aux assertions lorsqu'on tente d'ajouter ou de retirer une carte à un paquet.

```
bool Deck_isEmpty (Deck const * deck);
bool Deck_isFull  (Deck const * deck);
```

La fonction `Deck_appendCard()` ajoute une carte donnée à la fin d'un paquet non-plein. Inversement, `Deck_popCard()` retire une carte à la fin d'un paquet non-vide et retourne celle-ci.

```
void Deck_appendCard (Deck * deck, Card card);
Card Deck_popCard    (Deck * deck);
```

La fonction d'initialisation `Deck_initEmpty()` initialise un paquet à vide (sans cartes), tandis que `Deck_initComplete()` crée un jeu complet (tous les rangs de toutes les couleurs). Le paquet est trié en groupant les couleurs dans l'ordre décroissant (pique, coeur, carreau, trèfle), et à l'intérieur de celles-ci en rangeant les rangs dans l'ordre décroissant, de l'As au Deux, c'est-à-dire dans l'ordre : As Ks ... 3s 2s, Ah Kh ... 3h 2h, Ad Kd ... 3d 2d, Ac Kc ... 3c 2c.

```
void Deck_initEmpty    (Deck * deck);
void Deck_initComplete (Deck * deck);
```

Le prédicat `Deck_pipsEqual()` teste si un paquet `deck` est le même que celui décrit par une chaîne `pips` de symboles séparés par une espace. (Cette fonction facilite les assertions dans `DeckTest`).

```
bool Deck_pipsEqual (Deck const * deck, char const pips[]);
```

La fonction `Deck_printPips()` affiche sur le flux de sortie `file` les symboles d'un paquet `deck` séparés par une espace.

```
void Deck_printPips (Deck const * deck, FILE * file);
```

La fonction `Deck_appendPips()` ajoute à la fin d'un paquet `deck` les cartes décrites par une chaîne `pips` de symboles séparés par une espace, et `Deck_initFromPips()` initialise `deck` avec exactement cette série de cartes. (Ces fonctions facilitent la création de paquets dans `DeckTest`).

```
void Deck_appendPips   (Deck * deck, char const pips[]);
void Deck_initFromPips (Deck * deck, char const pips[]);
```

Le prédicat `Deck_indexIsValid()` teste si `index` est un index de carte valide pour le paquet `deck`. De même, `Deck_rangeIsValid()` teste la validité de la plage d'index de longueur `length` débutant à l'index `start`. (Ces fonction sont utilisées dans les assertions sécurisant les fonctions manipulant des index et des plages d'index).

```
bool Deck_indexIsValid (Deck const * deck, int index);
bool Deck_rangeIsValid (Deck const * deck, int start, int length);
```

La fonction `Deck_swapCardsAt()` permute les deux cartes d'index `index1` et `index2` dans un paquet `deck`. La fonction `Deck_shuffle()` mélange un paquet `deck` par l'algorithme de Fisher-Yates-Knuth : les index sont parcourus de gauche à droite, et on permute à chaque fois la carte de l'index courant `k` avec celle d'un index `>= k` choisi au hasard (et qui peut être `k` lui-même). On aura donc intérêt à écrire aussi la fonction auxiliaire `Random_intBetween()` qui tire un entier au hasard entre `left` et `right` inclus.

```
static int Random_intBetween (int left, int right);
void Deck_swapCardsAt (Deck * deck, int index1, int index2);
void Deck_shuffle (Deck * deck);
```

La fonction de distribution `Deck_dealCardsTo()` retire `cardCount` cartes du paquet `source` et les ajoute une à une à fin du paquet `target` dans l'ordre inverse. (Elle permet par exemple d'obtenir l'ensemble de 7 cartes dans laquelle on cherchera la meilleur main de 5 cartes).

```
void Deck_dealCardsTo (Deck * deck, int cardCount, Deck * target);
```

Les fonctions `Deck_sortBySuitFirst()` et `Deck_sortByRankFirst()` trient sur place un paquet `deck` en ordre décroissant, pour les ordres induits par les comparaisons `Card_compareByRankFirst()` et `Card_compareBySuitFirst()`, respectivement. Elles utilisent la fonction `qsort()` en combinaison avec deux wrappers sur ces fonctions de comparaison.

```
static int reverseCompareCardBySuitFirst (void const * data1, void const * data2);
static int reverseCompareCardByRankFirst (void const * data1, void const * data2);
void Deck_sortBySuitFirst (Deck * deck);
void Deck_sortByRankFirst (Deck * deck);
```

Les prédicats `Deck_isSortedByRankFirst()` et `Deck_isSortedBySuitFirst()` testent si un paquet `deck` est trié conformément aux deux fonctions de tris précédentes. (Ils servent aux assertions des fonctions d'extraction de main qui supposent le paquet trié selon un de ces critères).

```
bool Deck_isSortedByRankFirst (Deck const * deck);
bool Deck_isSortedBySuitFirst (Deck const * deck);
```

Les 5 fonctions de la famille `Deck_lengthOf...At()` retournent la longueur maximale d'un groupe de cartes situé à partir de l'index `start` dans un paquet `deck`. Les groupes détectés sont respectivement la sorte (cartes appariées), le flush (cartes assorties), le straight régulier (connecteurs réguliers), le straight-flush régulier (connecteurs réguliers assortis), les doublons (cartes identiques).

```
int Deck_lengthOfKindAt            (Deck const * deck, int start);
int Deck_lengthOfFlushAt           (Deck const * deck, int start);
int Deck_lengthOfRegularStraightAt (Deck const * deck, int start);
int Deck_lengthOfRegularStrFlushAt (Deck const * deck, int start);
int Deck_lengthOfEqualsAt          (Deck const * deck, int start);
```

Les 4 fonctions de la famille `Deck_startOfHighest...()` retournent l'index de départ du plus haut groupe de longueur au moins `length` dans un paquet `deck`. Les groupes détectés sont respectivement la sorte (cartes appariées) le flush (cartes assorties), le straight régulier (connecteurs réguliers), le straight-flush régulier (connecteurs réguliers assortis). Certaines de ces fonctions utilisent `Card_compareByArrayByRankOnly()` pour déterminer le plus haut groupe.

```
int Deck_startOfHighestKind            (Deck const * deck, int kindLength);
int Deck_startOfHighestFlush           (Deck const * deck, int flushLength);
int Deck_startOfHighestRegularStraight (Deck const * deck, int straightLength);
int Deck_startOfHighestRegularStrFlush (Deck const * deck, int strFlushLength);
```

La fonction `Deck_killRangeAt()` supprime de `deck` les `length` cartes situées à partir de l'index `start`. Les cartes suivantes sont décalées vers la gauche pour combler le trou. La fonction `Deck_copyRangeAt()` rajoute à la fin de `target` une copie des `length` cartes situées à partir de l'index `start` dans `deck`, dans le même ordre. (Utilisées ensemble, ces fonctions permettent l'extraction d'un groupe de cartes d'un paquet vers un autre.)

```
void Deck_killRangeAt (Deck       * deck, int start, int length);
void Deck_copyRangeAt (Deck const * deck, int start, int length, Deck * target);
```

La fonction `Deck_keepOnlyOneCardPerKind()` prend un paquet `deck` supposé déjà trié par rang d'abord, et ne garde que la première carte des groupes de cartes de même rang. La fonction `Deck_keepOnlyOneCardPerEqual()` fait de même avec un paquet supposé déjà trié par couleur d'abord, en ne gardant que la première carte des groupes de cartes identiques.

```
void Deck_keepOnlyOneCardPerKind  (Deck * deck);
void Deck_keepOnlyOneCardPerEqual (Deck * deck);
```

Le prédicat `Deck_isSortedByRankWithNoKind()` teste qu'un paquet `deck` est trié par rang d'abord sans deux cartes de même rang, et `Deck_isSortedBySuitFirstWithNoEqual()` teste qu'il est trié par couleur d'abord sans deux cartes identiques.

```
bool Deck_isSortedByRankWithNoKind       (Deck const * deck);
bool Deck_isSortedBySuitFirstWithNoEqual (Deck const * deck);
```

# 3 Le module `Hand`

Le module `Hand` fournit les fonctions suivantes :

La fonction `Hand_extractKind()` recherche dans le paquet `source` la plus grande sorte de longueur au moins `kindLength`, et l'en retire. Elle rajoute ensuite exactement `kindLength` cartes de cette sorte à la fin de `target` (la fonction retire donc éventuellement plus de cartes de `source` qu'elle n'en rajoute à `target`). Selon que `kindLength` vaut 1, 2, 3, ou 4, la fonction extrait le plus grand kicker, la plus grande paire, le plus grand brelan, ou le plus grand carré. La fonction suppose que le paquet `source` est déjà trié par rang d'abord : ainsi, les rangs identiques sont contigus avec les plus hauts rang en tête, et la sorte recherchée est celle la plus à gauche qui a une longueur au moins égale à `kindLength`. La fonction retourne `true` ssi elle a bien trouvé une sorte à extraire.

```
bool Hand_extractKind            (Deck * source, int kindLength, Deck * target);
```

Les fonctions `Hand_findHiCard()`, `Hand_findPair()`, `Hand_find2Pairs()`, `Hand_findTrips()`, `Hand_findBoat()`, et `Hand_findQuads()` recherchent la plus haute main de 5 cartes composée respectivement de 5 kickers, de 1 paire+3 kickers, de 2 paires+1 kicker, de 1 brelan+2 kickers, de 1 brelan+1 paire, et de 1 carré+1 kicker. Ces fonctions ne supposent pas que le paquet `source` est trié, et elle ne le modifient pas : il faut donc travailler sur une copie triée pour utiliser `Hand_extractKind()`. Elle ajoutent autant de cartes que possible à la fin `target` pour construire la main. Elles retournent `true` ssi toutes les 5 cartes de la main ont pu être ajoutées.

```
bool Hand_findHiCard   (Deck const * source, Deck * target);
bool Hand_findPair     (Deck const * source, Deck * target);
bool Hand_find2Pairs   (Deck const * source, Deck * target);
bool Hand_findTrips    (Deck const * source, Deck * target);
bool Hand_findBoat     (Deck const * source, Deck * target);
bool Hand_findQuads    (Deck const * source, Deck * target);
```

Le fonctions d'extraction suivantes sont les analogues de `Deck_extractKind()` pour l'extraction et la recherche d'un *straight* régulier, d'un wheel straight, d'un flush, d'un *straight-flush* régulier et d'un *wheel straight-flush* Les fonctions d'extraction de *straight* supposent que le paquet `source` est trié par rang d'abord sans cartes appariées. Les fonctions d'extraction de flush et *straight-flush* supposent que le paquet `source` est trié par couleur d'abord sans doublons.

```
bool Hand_extractRegularStraight (Deck * source, Deck * target);
bool Hand_extractWheelStraight   (Deck * source, Deck * target);
bool Hand_extractFlush           (Deck * source, Deck * target);
bool Hand_extractRegularStrFlush (Deck * source, Deck * target);
bool Hand_extractWheelStrFlush   (Deck * source, Deck * target);
```

Les fonctions de recherche associées ne modifient pas le paquet `source` et ne font pas d'hypothèse sur son tri : il faut donc travailler sur une copie triée pour utiliser les fonctions d'extraction.

```
bool Hand_findStraight (Deck const * source, Deck * target);
bool Hand_findFlush    (Deck const * source, Deck * target);
bool Hand_findStrFlush (Deck const * source, Deck * target);
```

Enfin, la fonction `Deck_findBest()` trouve la meilleure main dans `source` et la place dans `target`. La fonction retourne la nature de la main, de type `Hand`.

```
Hand Hand_findBest (Deck const * source, Deck * target);
```