Le jeu de Mastermind

I. Introduction

Le jeu de *Mastermind* fut inventé vers 1970 par Mordecai Meirowitz. Dans sa version d'origine, il faut trouver un *code* constitué d'une suite de 4 pions dont les couleurs sont choisies parmi 6 différentes. Le code est choisi par l'un des joueurs, le *codeur*, et l'autre joueur, le *décodeur*, doit le déterminer en faisant des propositions de code jusqu'à trouver le bon. À chaque proposition, le codeur indique au décodeur le nombre de pions bien placés, et le nombre de pions mal placés.

Par exemple, si le code à deviner (on utilise ici des chiffres plutôt que des couleurs) est :

```
1 2 1 3
```

et que le décodeur propose le code :

```
1 2 3 1
```

alors le nombre de pions bien placés est 2 (des deux pions les plus à gauche) et le nombre de pions mal placés est aussi 2 (il s'agit de l'autre `1` et du `3`).

Question 1 Combien y a-t-il de codes différents possibles ?

II. Bien placés, mal placés

Dans cette partie, on veut écrire des fonctions qui, étant donné un code et une proposition de code, calculent le nombre de pions bien placés et de pions mal placés de la proposition. Les codes seront représentés sous forme de listes d'entier, et les couleurs seront codées par des entiers de `1` à `6`. Ainsi, le code précédent est représenté par `[1, 2, 1, 3]` et la proposition par `[1, 2, 3, 1]`.

Question 2 Écrire une fonction `occurences` qui, étant donné une liste d'entiers `l` et un entier `n`, retourne le nombre d'occurences de la valeur `n` dans `l` (autrement dit, le nombre de fois que la valeur apparaît dans la liste). Cette fonction pourra avoir la forme d'un parcours complet de tableau.

On veut, par exemple :

```
>>> l = [1, 2, 4, 3, 5, 1, 2, 1]
>>> occurences(l, 1)
3
>>> occurences(l, 2)
2
>>> occurences(l, 6)
0
```

Question 3 Écrire une fonction `bien_places` qui, étant donné deux listes `l1` et `l2` de même longueur, indique le nombre de pions *bien placés*, c'est-à-dire le nombre de positions où les deux listes ont le même élément. À nouveau, on privilégiera une parcours de tableau.

Par exemple :

```
>>> bien_places([1, 2, 1, 1, 3], [2, 2, 1, 2, 4])
2
>>> bien_places([1, 2, 3, 4, 5, 1, 2], [1, 2, 3, 4, 2, 2, 1])
4
```

Si, étant donné un code c et une proposition de code p, si l'on note $|c|_i$ (resp. $|p|_i$) le nombre d'occurences de la couleur i dans c (resp. dans p) et b le nombre de pions bien placées, alors le nombre m de pions à la mauvaise place est égal à :

$$m = \left(\sum_{i=1}^{6} \min(|c|_i, |p|_i)\right) - b$$

Question 4 En déduire une fonction `mal_places` qui, étant donné deux listes `l1` et `l2`, indique le nombre de pions mal placés. Elle pourra utiliser les fonctions déjà programmées.

Par exemple,

```
>>> mal_places([1, 2, 2, 1, 3], [1, 2, 1, 4, 2])
2
```

III. L'ordinateur est le codeur

Maintenant que nous avons les fonctions de base, nous allons programmer le cas où c'est l'ordinateur qui fait deviner un code à l'utilisateur.

Question 5 En utilisant la fonction `randint` du module `random` (que vous devrez importer), écrire une fonction `creer_code` qui ne prend pas d'argument et retourne un code engendré aléatoirement.

On peut obtenir, par exemple :

```
>>> creer_code()
[2, 5, 6, 2]
>>> creer_code()
[2, 3, 5, 5]
>>> creer_code()
[3, 2, 2, 3]
>>> creer_code()
[4, 1, 2, 6]
```

Question 6 Étant donné un code et une proposition de code, à quelle condition sur les nombres de pions bien placés et mal placés le décodeur a-t-il gagné ?

Voici maintenant une fonction permettant à l'utilisateur d'entrer une proposition sous la forme de 4 chiffres séparés d'espaces :

```
def demande_proposition():
    p = input("Proposition : ")
    return [int(x) for x in p.split(" ")]
```

Le comportement est alors le suivant :

```
>>> prop = demande_proposition()
Proposition : 3 2 4 1
>>> prop
[3, 2, 4, 1]
```

Question 7 Écrire une fonction `jeu` sans argument qui simule un jeu de *Mastermind* avec l'ordinateur comme codeur. La fonction pourra se structurer ainsi :

1. Choisir aléatoirement un code ;
2. *Tant que* le code n'a pas été trouvé,
   - ⋆ demander une proposition,
   - ⋆ verifier si elle est correcte,
   - ⋆ et dans le cas contraire, afficher les nombres de pions bien et mal placés, et recommencer.

Question 8 Modifier la fonction précédente pour qu'elle retourne à la fin le nombre d'essais nécessaires pour trouver le code.

Question 9 Essayez de modifier la fonction précédente pour que le joueur perde s'il n'a pas trouvé au bout de 10 essais.

IV. L'ordinateur est le décodeur

Dans cette partie, plus difficile et réservée aux plus courageux, nous allons tenter de programmer un ensemble de fonctions pour que l'ordinateur joue cette fois le rôle du décodeur et arrive à trouver une combinaison proposée par l'utilisateur en un nombre raisonnable d'essais.

1. Présentation de l'algorithme

Pour cela, nous allons utiliser un algorithme très simple : l'ordinateur va essayer tous les codes possibles dans l'ordre, et ne proposer à l'utilisateur que ceux qui sont compatibles avec les informations précédentes.

Exemple détaillé Supposons que l'on veut faire trouver le code `3 2 2 4` à l'ordinateur. L'ordinateur propose `1 1 1 1`, et on lui répond qu'il y en a 0 bien placés et 0 mal placés (autrement dit, l'ordinateur sait qu'il n'y a pas de `1`). Le code suivant, `1 1 1 2`, n'est pas compatible avec ces informations, ni `1 1 1 3`

Le code suivant qu'il propose est donc le `2 2 2 2`, et on lui répond qu'il y a 2 pions bien placés et 0 mal placés (autrement dit, il y a deux `2` dans le code, mais l'ordinateur ne va pas utiliser cette information ainsi).

En continuant d'énumérer les codes possibles, le prochain code compatible avec l'information disponible (qui indique qu'il n'y a pas de `1` et qu'il y a deux `2`) est le code `2 2 3 3`. On lui répond qu'il y a 1 pion bien placé et 2 pions mal placés.

À ce moment, les informations reçues par l'ordinateur sont que :

- ⋆ vis-à-vis de `1 1 1 1`, il y a 0 bien placés et 0 mal placés ;
- ⋆ vis-à-vis de `2 2 2 2`, il y a 2 bien placés et 0 mal placés ;
- ⋆ vis-à-vis de `2 2 3 3`, il y a 1 bien placés et 2 mal placés.

2. Représentation des données et fonctions utiles

Chacun des essais précédents peut se représenter à l'aide d'un triplet

```
(proposition, bien, mal)
```

où l'on stocke la *proposition*, c'est-à-dire le code proposé, ainsi que les réponses obtenues : le nombre de pions *bien placés* et ceux *mal placés*.

Question 10 Écrire une fonction `test_essai` qui, étant donné un code et un essai (sous la forme du triplet précédent) indique à l'aide d'un booléen si le code donné a les bons nombres de pions bien placés et mal placés vis-à-vis de la proposition.

Il pourra avoir la structure suivante :

```
def test_essai(code, essai) :
    proposition, bien, mal = essai
    # a compléter
```

Un exemple de fonctionnement est le suivant :

```
>>> test_essai([1, 1, 2, 1], ([1, 1, 1, 1], 3, 0))
True
>>> test_essai([1, 1, 2, 1], ([1, 2, 3, 1], 2, 2))
False
```

Nous allons maintenant stocker ces informations sous la forme d'une liste, chaque élément étant un triplet proposition/bien placés/mal placés. Ainsi, les différents essais de l'exemple détaillé seront codées par la liste :

```
[([1, 1, 1, 1], 0, 0),
 ([2, 2, 2, 2], 2, 0),
 ([2, 2, 3, 3], 1, 2)]
```

Question 11 Écrire une fonction `test_code` qui, étant donné un code `c` et une liste `l` du format précédent, indique si le code `c` est compatible avec les informations contenues dans la liste `l`, autrement di si le code `c` donne les nombres corrects de pions bien et mal placés pour chaque proposition contenue dans `l`.

Ainsi, on doit avoir :

```
>>> l = [([1, 1, 1, 1], 0, 0)]
>>> test_code([1, 1, 2, 1], l)
False
>>> test_code([2, 2, 2, 2], l)
True
>>> l = [([1, 1, 1, 1], 0, 0),
...      ([2, 2, 2, 2], 2, 0),
...      ([2, 2, 3, 3], 1, 2)]
>>> test_code([2, 3, 2, 3], l)
False
>>> test_code([2, 3, 2, 4], l)
True
```

Question 12 Écrire une fonction `jeu_2` sans argument qui implémente l'algorithme précédent. On pourra écrire des fonctions auxilliaires si nécessaire, et il est recommandé de se poser les questions suivantes :

- ⋆ Quelle est la liste correspondant à l'information disponible au début du jeu ?
- ⋆ Comment énumérer tous les codes possibles ?
- ⋆ Si une proposition de code est fausse, elle permet d'acquérir de nouvelles informations. Comment cela se traduit-il ?

3. Quelques questions supplémentaires

Si vous en voulez encore, voici quelques idées à explorer.

Question 13 Avec l'algorithme précédent, il faut en moyenne 5.76 essais pour trouver un code. Pourriez-vous retrouver ce résultat ? Une manière de procéder et d'appliquer l'algorithme à tous les codes possibles, d'additionner les nombres de coups nécessaires et de diviser par le nombre de code. Simple, non ?

Question 14 Dans le même ordre d'idée, quel est le nombre maximal de coups nécessaires pour trouver un code ?

Question 15 En fait, énumérer les codes possibles dans l'ordre n'est pas la meilleure idée. Il est bien plus efficace, lors de la recherche, d'énumérer les codes de manière aléatoire. Pour cela, on peut créer la liste de tous les codes, la mélanger avec la fonction `random.shuffle` et de parcourir la liste mélangée pour chercher un code. Sauriez-vous programmer cela ?

Question 16 Il y a bien sûr des fluctuations dûes à l'ordre aléatoire utilisé pour énumérer les codes, mais on passe à environ 4.64 essais en moyenne. Pourriez-vous retrouver cela ?