Riez Esteban
Biret Margo
Groupe 8

Projet Mastermind - Informatique S4

Question 1

Voici le test réalisé pour la fonction **evaluation**:

```
print("RBBG avec RBVG:",evaluation('RBBG','RBVG'))
print("JBVG avec ORMN:",evaluation( 'JBVG','ORMN'))
```

Comme voulu, la fonction nous renvoie un couple de deux entiers:

```
RBBG avec RBVG: (3, 0)
JBVG avec ORMN: (0, 0)
```

Question 2

Pour réaliser le test de cette nouvelle fonction **codemaker** il faut d'abord lancer **init,** puis faire l'évaluation:

```
init()
print("solution:",solution)
print("evaluation pour OJVG:",codemaker('OJVG'))
print("evaluation pour MNRB:",codemaker('OJJG'))
```

La fonction codemaker utilise **evaluation** que nous avons créé à la question précédente. Elle renvoie également un couple de deux entiers:

```
Reloaded modules: common
solution: VVRR
evaluation pour OJVG: (0, 1)
evaluation pour MNRB: (0, 0)
```

Question 3

Nous allons tout d'abord calculer la valeur théorique de l'espérance du nombre d'essais réalisés par la version 0 de codebreaker.
Pour cela, nous calculons le nombre total de possibilités de combinaison de quatre éléments avec huit couleurs. Puisque l'ordre est important et que les répétitions sont autorisées on a 8^4 possibilités. La probabilité de trouver en essai est donc de $\frac{1}{8^4}$.

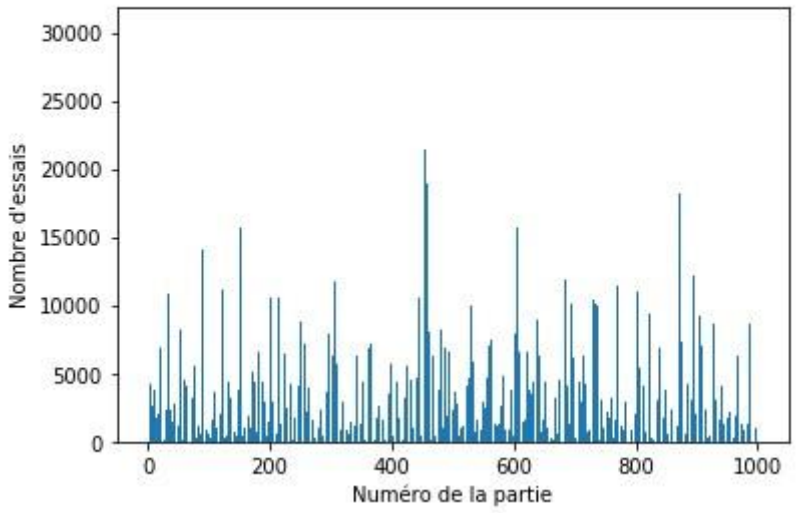
Nous avons ici une expérience de Bernoulli (avec comme succès "la combinaison est la bonne" et comme échec "la combinaison n'est pas la bonne"). Cette expérience de Bernoulli est répétée indépendamment jusqu'à obtenir le premier

succès. Nous avons donc une loi géométrique de paramètre $p = \frac{1}{8^4}$. Or l'espérance d'une loi géométrique est $E(X) = \frac{1}{p} = \frac{1}{\frac{1}{8^4}} = 8^4 = 4096$.

Nous avons ensuite effectué 1000 parties de mastermind et nous avons tracé graphiquement pour chaque partie le nombre d'essais réalisés. Afin de tracer le graphique il nous a fallu créer une fonction qui effectuait les 1000 parties et qui enregistrait le nombre d'essais dans un tableau:

```
def plusieurs_play(codemaker,codebreaker,nbr):
    """
    Cette fonction sert à faire jouer codemaker contre codebreaker un nombre de fois donné.
    Elle enregistre ensuite le nombre d'essai réalisé pour chaque partie.
    Les arguments sont le codemaker puis le codebreaker que l'on veut faire jouer
    suivi du nombre de parties que l'on veut réaliser.
    """
    donnees=[]
    for i in range(nbr):
        #à chaque partie on ajoute le nombre d'essai dans donnees
        donnees.append(play(codemaker, codebreaker))
    return donnees
```

Nous avons ensuite utilisé *matplotlib* et ses fonctions associées pour créer notre graphique. Voici l'histogramme obtenu:



Nous pouvons observer que d'une partie à l'autre le nombre d'essai varie grandement: il peut être très grand (plus de 20000 ici) comme très petit (il y en a sous les 1250).

Afin de pouvoir comparer notre valeur théorique à celle expérimentale sur les 1000 essais, nous avons également calculé la moyenne à l'aide de s*tatistiques* depuis Python:

```
print("La moyenne est de: ",statistics.mean(resultat))
```

```
La moyenne est de:  3939.98
```

Nous pouvons observer que la moyenne expérimentale est assez proche de la moyenne théorique (3940 contre 4096).

Question 4

Comme précédemment nous allons calculer l'espérance du nombre d'essais réalisés pour une partie avec codebreaker1.

La probabilité de trouver au premier essai est $\frac{1}{4096}$. Donc la probabilité de ne pas trouver au premier essai est de $\frac{4095}{4096}$.

La probabilité de trouver au deuxième essai vaut $\frac{1}{4095} \times \frac{4095}{4096} = \frac{1}{4096}$ car si nous trouvons au deuxième essai c'est que nous n'avons pas trouvé au premier.

Ainsi, la probabilité de trouver au n-ième essai est de $\frac{1}{4096-n} \times \frac{4096-n}{4096} = \frac{1}{4096}$.

Calculons ensuite l'espérance de cette probabilité, c'est-à-dire le nombre moyen d'essais par partie: $E(X) = \sum_{n=1}^{4096} \frac{1}{4096} \times n$.

Pour effectuer ce calcul, nous avons utilisé un court code:

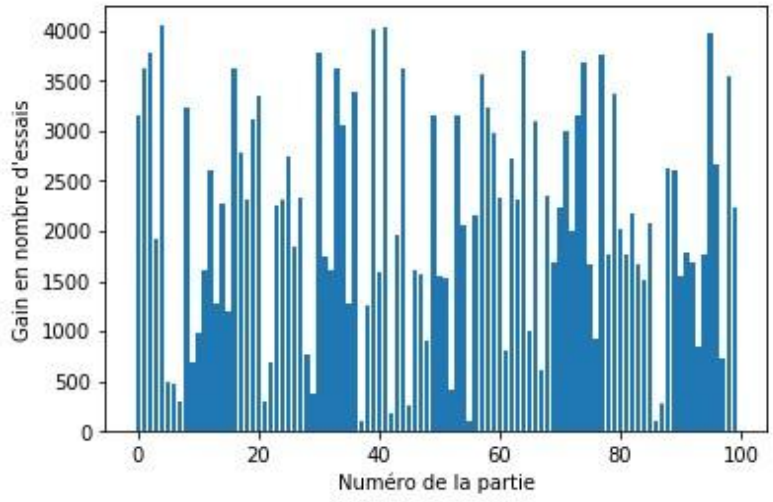
```
def somme():
    s=1
    #pour i allant de 1 à 4096
    for i in range (1,4096):
        #on ajoute à s i fois la probabilité
        s+=(1/4096)*i
    return s

#on fait tourner le programme et on le rentre dans une variable
p=somme()
#on l'affiche
print(p)
```

Ce code renvoie la valeur de 2048,5.

Afin d'avoir le gain en nombre d'essai nous allons soustraire cette nouvelle espérance à l'ancienne: théoriquement nous avons $G = 4096 - 2048,5 = 2047,5$.

Afin de voir cela graphiquement nous avons utilisé une fonction du même type que **plusieurs_play** mais cette fois-ci nous avons rentré directement le gain (c'est-à-dire 4096-le nombre d'essais de la partie) dans le tableau de données. Voici le graphique obtenu:



Ici nous pouvons voir que le gain en nombre d'essais varie grandement d'une partie à l'autre: il peut y avoir environ 4000 essais de gagnés comme 250.

Comme précédemment nous avons afficher la moyenne du tableau de données (ici contenant le gain pour chaque partie) afin de pouvoir le comparer à notre valeur théorique. Nous avons obtenu cela: `Gain moyen de:  2077.28` .

Ainsi nous voyons que notre gain expérimental est proche de celui théorique (2077,28 contre 2047,5).

Question 5

Dans cette question nous avons été confrontés à un problème avec le set. En effet la fonction nous renvoyait un set du type : {('J','V','B','R'),('G','G','M','N')...}.
Ainsi, lorsqu'il fallait choisir une combinaison avec la fonction **choices** cela ne fonctionnait pas. Pour régler ce soucis et avoir un set du type: {('JVBR'),('GGMN')...} nous avons créé une fonction intermédiaire **chaine** que **donner_possibles** utilise:

```
def chaine (q):
    """
    Cette fonction permet de prendre un tuple de 4 lettres pour le mettre sous
    la forme d'une seule chaîne de 4 caractères.
    L'argument est donc un quadruplet du type ('J','B','B','G') et renvoie
    une chaîne de caractère du type 'JBBG'.
    """
    combinaison='' #on initialise la variable dans laquelle on mettra notre chaîne de caractères
    #pour chaque élément de l'argument
    for i in q:
        combinaison+=i #on l'ajoute à la chaîne de caractère
    return combinaison
```

Dans la fonction **donner_possibles** nous avons également utilisé la fonction **maj_possible** de la question d'après. En effet, la fonction **donner_possible** sert juste à créer le set une première fois avec toutes les combinaisons possibles. Pour

cela nous avons utilisé *itertools.product* déjà présents dans python car cela permet de créer toutes les combinaisons possibles avec l'ordre qui importe et des répétitions possibles. Les arguments de cette fonction sont les caractères composants les combinaisons et la longueur de la combinaison que l'on souhaite (ici COLORS et LENGTH).
L'appel à **maj_possibles** permet de supprimer les combinaisons non compatibles avec l'essai proposé et l'évaluation renvoyée.

Voici un exemple du set renvoyé par **donner_possibles** avec l'essai JRGV et l'évaluation (2,1) correspondante.

```
In [27]: print(common.donner_possibles('JRGV',
(2,1)))
{'GJGV', 'ORGJ', 'JRVN', 'JJGR', 'JJRV', 'JNGR',
'GRVV', 'JVGM', 'VRJV', 'VRGR', 'GROV', 'GRRV',
'JGBV', 'BJGV', 'NRJV', 'JVRV', 'GRGJ', 'JRVB',
'JBRV', 'JVGO', 'JRMG', 'JMRV', 'JOGR', 'VRGB',
'VJGV', 'ORJV', 'RMGV', 'GRNV', 'GRMV', 'RRJV',
'RBGV', 'JBGR', 'MRJV', 'JRVJ', 'JGGR', 'NJGV',
'JVGN', 'JRVO', 'RRGJ', 'OJGV', 'JVGJ', 'JVGG',
'JRVM', 'VRGO', 'JGJV', 'JGNV', 'JNRV', 'JVGB',
'GRBV', 'JORV', 'JROG', 'BRJV', 'JRJG', 'JRBG',
'JRVR', 'RNGV', 'JRRG', 'NRGJ', 'RGGV', 'JGMV',
'JGOV', 'ROGV', 'VRGG', 'JRNG', 'JGVV', 'VRGN',
'JMGR', 'MRGJ', 'RVGV', 'BRGJ', 'VRGM', 'MJGV'}
```

Question 6

Afin de respecter la consigne de cette question et modifier directement le set donné en argument, nous avons choisi de créer une copie afin d'itérer sur les éléments de la copie tout en modifiant le set de l'argument. En effet, avec un set il n'est pas possible d'itérer sur les index.

Question 7

Pour cette fonction il nous a fallu distinguer trois cas:

-Si codebreaker n'a encore jamais rien proposé. Dans ce cas il n'y pas de combinaison impossible: codebreaker en choisit une parmis les 4096.

-Si codebreaker a fait une unique proposition. Dans ce cas, le set avec les combinaisons possibles n'a pas encore été créé. Nous utilisons donc **donner_possibles**.

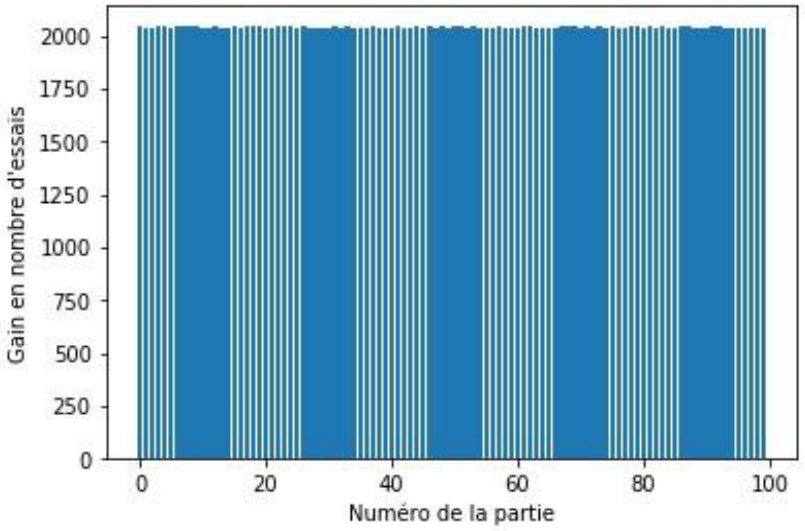
-Si codebreaker a fait plus d'une proposition. Dans ce cas, le set de combinaisons possibles existe et il faut le mettre à jour avec le nouvel essai et l'évaluation associée. Pour cela nous utilisons **maj_possibles.**

Nous avons tout d'abord voulu voir l'espérance de ce nouveau **codebreaker** avec la fonction **plusieurs_play** :

```
print("Nombre d'essais moyen avec codebreaker2: ", statistics.mean(plusieurs_play(codemaker1,codebreaker2,100)))
```

```
Nombre d'essais moyen avec codebreaker2:  5.57
```

Nous avons ensuite comparé graphiquement et numériquement le gain par rapport à codebreaker1. Pour cela nous avons comme précédemment rentré directement le gain dans notre tableau de données (c'est-à-dire l'espérance de codebreaker1 qui vaut 2048,5 moins le nombre d'essai pour chaque partie), puis nous avons affiché tout cela graphiquement:



Nous remarquons ici que le gain moyen est presque constant: on peut en conclure que le nombre d'essais par partie réalisé par codebreaker2 est relativement constant également.

```
Gain moyen:  2042.93
```

Enfin nous pouvons observer que ce résultat se rapproche de 2048-5,57=2042,43 (en prenant en compte une moyenne de 5,57 essais avec codebreaker2).

<u>Question 8</u>

Pour cette question nous avons décidé de changer la solution seulement si cela donnait moins d'informations au codebreaker. Afin de respecter la contrainte de cohérence des anciennes évaluations nous avons d'abord voulu utilisé combi_possibles que codebreaker2 met à jour à chaque essai. Cela nous garantissait que la nouvelle solution était en accord avec les anciennes évaluations. Malheureusement on s'est rendu compte que même en utilisant combi_possibles comme variable globale, codemaker2 ne la reconnaissait pas. Cela est dû au fait qu'elle ne soit pas dans le même fichier.

Pour surmonter ce problème nous avons eu deux idées: la première était d'importer combi_possibles depuis codebreaker2, l'autre était de créer un autre set que codemaker2 mettrait lui-même à jour. Nous avons opté pour la deuxième option permettant de rendre totalement indépendant le jeu de codebreaker et codemaker. Cela permet aussi de jouer contre codemaker2 en tant qu'humain.

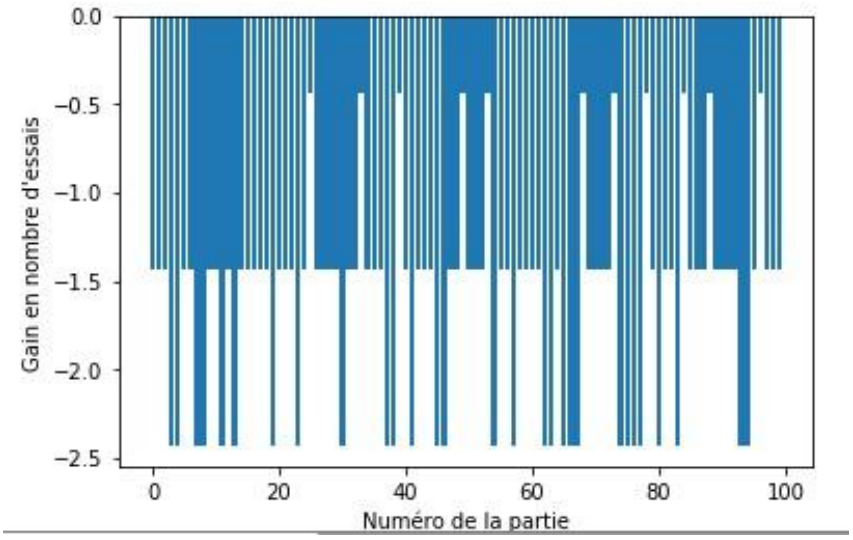
Nous avons donc utilisé les fonctions **donner_possibles** et **maj_possibles** et il nous a également fallu distinguer deux cas: si c'est la première proposition de codebreaker alors pas la peine de changer la solution car nous n'avons jamais fait d'évaluation avec. De plus il faut créer combi_compatibles avec **donner_possibles** afin de l'utiliser ensuite avec **maj_possibles** pour les autres propositions de codebreaker.
Enfin nous avons fait en sorte que codemaker choisisse toujours une solution différente de l'essai fait par codebreaker (sauf au dernier essai) pour ne pas le laisser gagner trop rapidement.

Voici le nombre d'essais moyen réalisés pour chaque partie avec codemaker2:

```
Nombre d'essais moyen avec codebreaker2 et
codemaker2:  7.3
```

Nous pouvons voir que le nombre d'essais par partie est supérieur lorsque codemaker triche (5,57 contre 7,3).

Voici un graphique du gain obtenu entre les deux versions du codemaker:



Nous pouvons remarquer que dans tous les cas le gain est négatif: cela veut dire que lorsque codebreaker triche le nombre d'essais est forcément plus grand donc la triche implique une partie plus longue.

```
Gain moyen avec codemaker2:  -1.6099999999999997
```

Nous pouvons aussi le voir avec le gain moyen qui est également négatif: nous pouvons en conclure que le codebreaker a besoin de plus d'essais pour trouver la solution lorsque codemaker triche.

Question 11

Pour cette question nous avons dû créer un fichier texte dans la fonction **play_log**. Il fallait que ce fichier texte prenne le nom donné en argument de la fonction: c'est le point sur lequel nous avons cherché le plus longtemps car nous ne savions pas comment le mettre dans la syntaxe *open("nom.txt","a").* Nous avons finalement créé une variable fichier dans laquelle nous implémentons le string "nom.txt" (qui varie en fonction de l'argument nom).
Afin de respecter la syntaxe exacte donnée dans *log0.txt* il a fallu être précis sur les sauts de ligne et les espaces.

Voici un exemple de fichier que nous renvoie la fonction avec codebreaker2 et codemaker2:

```
RROV
0,1
MVBN
0,2
NGMO
1,1
BNNO
0,1
VGNG
0,1
GRMB
0,2
NJRM
1,3
NMJR
4,0
```

Question 12

Afin de réaliser la fonction **check_codemaker** de cette question, nous avons dû récupérer les informations présentes dans le fichier texte log. Le plus dur a été de mettre les informations sous la bonne forme pour qu'elles soient ensuite utilisées par les autres fonctions que nous avons créées lors de ce projet.

Pour faire **check_codemaker** nous avons eu une première idée: pour chaque essai et son évaluation du fichier log nous voulions créer et utiliser un set de combinaisons possibles à l'aide de **maj_possibles** et **donner_possibles**. Malheureusement nous nous sommes ensuite demandé comment interpréter l'information que ce set nous donnait.
Nous avons alors changé de stratégie et nous avons eu une deuxième idée: en prenant le dernier essai, qui est donc la solution, nous avons vérifié que pour tous les autres essais l'évaluation correspondait bien avec celle du fichier log. Si l'évaluation pour l'un des essais avec la solution ne correspond pas à celle du fichier, c'est que codemaker a changé de solution pendant le jeu.

Question 13

Après avoir fini ce projet Mastermind, nous pensons pouvoir dire que nous y avons passé environ une trentaine d'heures (séances avec les professeurs incluses).
Ce projet nous a posé quelques problèmes informatiques car nous avons été confronté à des nouvelles notions (par exemple les set que nous n'avions jamais utilisé en cours) ainsi que des problèmes théoriques (le meilleur exemple serait la

question 4 qui demandait un calcul théorique de probabilité et d'espérance assez complexe, sur lequel nous avons passé plus de 3h). Le travail en groupe est donc essentiel pour régler ces problèmes car chacun possède des compétences différentes.
L'organisation de notre binôme s'est principalement appuyée sur les compétences de chacun: Margo a codé la plupart des questions grâce à ses aisances en programmation tandis qu'Esteban s'est attelé à la partie mathématique. Nous réfléchissions ensemble aux idées et manières de faire, Esteban a effectué les calculs mathématiques et Margo s'est occupée de traduire tout cela en code pour mettre en forme le projet.
Le travail en groupe a donc été un gain de temps puisque nous avons réparti les tâches. La communication entre nous deux a été bonne ce qui a également facilité l'avancée du projet. La programmation pour répondre aux questions venant principalement d'une seule personne, nous n'avons pas eu la nécessité de partager le code. Nous nous sommes retrouvés plusieurs fois afin de coder ensemble sur le même ordinateur.