



Compte-Rendu - TP Informatique **Mastermind**

Quentin Zoppis, Romain Peyremorte, Yael Gras

25 Avril 2021

Table des matières

1	Introduction	2
2	Le code	2
2.1	Fonction evaluation	2
2.2	Codemaker1	3
2.3	Codebreaker1	4
2.4	Fonction donner_possibles	4
2.5	Fonction maj_possibles	5
2.6	Codebreaker2	5
2.7	Codemaker2	6
2.8	Codebreaker3	7
2.9	Fonction play_log	8
2.10	Check_codemaker	9
3	Questions	10
3.1	Question 3 - Analyse de la performance du codebreaker0	10
3.2	Question 4 - Analyse de la performance du codebreaker1	11
3.3	Question 7 - Analyse de la performance du codebreaker2	13
3.4	Question 8 - Analyse de la performance du codemaker2	14
3.5	Question 9 - Utilité de tester une combinaison impossible	16
3.6	Question 10 - Analyse de la performance du codebreaker3	17
4	Conclusions - Question 13	20

1 Introduction

Ce programme aura pour objectif de faire jouer automatiquement un codebreaker et un codemaker l'un contre l'autre ou l'un contre un joueur. Nous avons donc codé différentes fonctions pour améliorer l'intelligence artificielle. Nous avons deux versions du codemaker et trois versions du codebreaker. Nous comparerons ces différentes versions grâce à différents graphiques. Ce compte-rendu sera articulé en deux grandes parties. La première expliquera succinctement les différentes fonctions que l'on a dû coder, un exemple de tests que l'on a produit et possiblement une critique de la fonction. La deuxième répondra aux différentes questions du sujet, elle comportera donc les graphiques et les calculs théoriques.

2 Le code

2.1 Fonction evaluation

Fonctionnement :

Le but de cette fonction est de déterminer le nombre de plots correctement placés (*red*) et celui de plots mal placés (*white*) dans une combinaison par rapport à une combinaison de référence. Elle prend en entrée la combinaison à évaluer et la combinaison de référence, et retourne sous forme d'un tuple *red* et *white*.

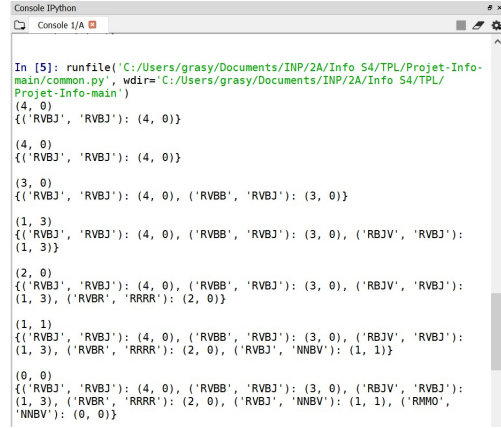
Pour enregistrer les couleurs bien placées, on parcourt une première fois la combinaison et la référence et nous comptons les couleurs bien placées, nous les supprimons au fur et à mesure des deux listes pour éviter de les recompter par la suite. Puis nous reparcourons la combinaison et la référence pour compter les couleurs mal placées.

Afin de ne pas refaire ces étapes de calcul plusieurs fois pour une même association combinaison/référence, on stocke dans le dictionnaire global *eval_combi_solu* nos évaluations avec comme clé cette association sous forme de tuple. À chaque lancement de la fonction, on regarde tout d'abord si l'évaluation est déjà dans notre dictionnaire, dans ce cas on retourne cette évaluation, sinon on calcule *red* et *white* comme décrit précédemment, on stocke cette évaluation dans notre dictionnaire, puis on la retourne.

Tests :

```
print(evaluation('RVBJ','RVBJ'))
print(eval_combi_solu)
print()
print(evaluation('RVBJ','RVBJ'))
print(eval_combi_solu)
print()
print(evaluation('RVBB','RVBJ'))
print(eval_combi_solu)
print()
print(evaluation('RBJV','RVBJ'))
print(eval_combi_solu)
print()
print(evaluation('RVBR','RRRR'))
print(eval_combi_solu)
print()
print(evaluation('RVBJ','NNBV'))
print(eval_combi_solu)
print()
print(evaluation('MMMO','NNBV'))
print(eval_combi_solu)
print()
```

FIGURE 1 – Programme test de la fonction évaluation



```
In [5]: runfile('C:/Users/grasy/Documents/INP/2A/Info S4/TPL/Projet-Info-
main/common.py', wdir='C:/Users/grasy/Documents/INP/2A/Info S4/TPL/
Projet-Info-main')
(4, 0)
{'RVBJ', 'RVBJ': (4, 0)}
(4, 0)
{'RVBJ', 'RVBJ': (4, 0)}
(3, 0)
{'RVBJ', 'RVBJ': (4, 0), ('RVBB', 'RVBJ'): (3, 0)}
(1, 3)
{'RVBJ', 'RVBJ': (4, 0), ('RVBB', 'RVBJ'): (3, 0), ('RBJV', 'RVBJ'):
(1, 3)}
(2, 0)
{'RVBJ', 'RVBJ': (4, 0), ('RVBB', 'RVBJ'): (3, 0), ('RBJV', 'RVBJ'):
(1, 3), ('RVBR', 'RRRR'): (2, 0)}
(1, 1)
{'RVBJ', 'RVBJ': (4, 0), ('RVBB', 'RVBJ'): (3, 0), ('RBJV', 'RVBJ'):
(1, 3), ('RVBR', 'RRRR'): (2, 0), ('RVBJ', 'NNBV'): (1, 1)}
(0, 0)
{'RVBJ', 'RVBJ': (4, 0), ('RVBB', 'RVBJ'): (3, 0), ('RBJV', 'RVBJ'):
(1, 3), ('RVBR', 'RRRR'): (2, 0), ('RVBJ', 'NNBV'): (1, 1), ('MMMO',
'NNBV'): (0, 0)}
```

FIGURE 2 – Console du test de la fonction évaluation

Avec ces tests, qui ne sont qu'une partie des tests faits, nous remarquons que l'évaluation est correcte pour toutes les combinaisons et références proposées. De plus, le dictionnaire se remplit bien des nouvelles combinaisons / références avec les évaluations faites.

2.2 Codemaker1

Fonctionnement :

Cette fonction est la version classique du codemaker, c'est-à-dire qu'en début de partie, il va choisir une solution et la garder tout au long de la partie pour renvoyer l'évaluation de la combinaison proposée par le codebreaker.

Tests :

```
init()
print('solution : ' + str(solution))
print('RVBJ : ' + str(codemaker('RVBJ')))
print('MMNO : ' + str(codemaker('MMNO')))
print('GGGG : ' + str(codemaker('GGGG')))
print()

init()
print('solution : ' + str(solution))
print('RVBJ : ' + str(codemaker('RVBJ')))
print('MMNO : ' + str(codemaker('MMNO')))
print('GGGG : ' + str(codemaker('GGGG')))
print()

init()
print('solution : ' + str(solution))
print('RVBJ : ' + str(codemaker('RVBJ')))
print('MMNO : ' + str(codemaker('MMNO')))
print('GGGG : ' + str(codemaker('GGGG')))
print()
```

FIGURE 3 – Programme test du fichier codemaker1

```
solution : VJBJ
RVBJ : (2, 1)
MMNO : (0, 0)
GGGG : (0, 0)
```

```
solution : JVGQ
RVBJ : (1, 1)
MMNO : (1, 0)
GGGG : (1, 0)
```

```
solution : RMGO
RVBJ : (1, 0)
MMNO : (2, 0)
GGGG : (1, 0)
```

FIGURE 4 – Console du test du fichier codemaker1

Ces tests nous permettent de vérifier que la fonction *init* génère bien une solution aléatoire et que la fonction *codemaker* réalise correctement l'évaluation d'une combinaison testée par rapport à la solution.

2.3 Codebreaker1

Fonctionnement :

Ce codebreaker enregistre les combinaisons qu'il a déjà proposées de manière à ne proposer que les solutions qu'il n'a pas encore testées. Pour cela, nous créons la liste de toutes les possibilités pour le nombre de couleurs et la longueur possible puis nous choisissons au hasard parmi cette liste puis nous enlevons de cette liste la combinaison choisie.

Tests :

global combinaisons_possibles

```
init()
print('Il y a ' + str(len(combinaisons_possibles))
      + ' combinaisons possibles à ce stade.')
print('Combinaison testée : ' + str(codebreaker((0,0))))
print('Il y a ' + str(len(combinaisons_possibles))
      + ' combinaisons possibles à ce stade.')
print('Combinaison testée : ' + str(codebreaker((0,0))))
print('Il y a ' + str(len(combinaisons_possibles))
      + ' combinaisons possibles à ce stade.')
print('Combinaison testée : ' + str(codebreaker((0,0))))
print('Il y a ' + str(len(combinaisons_possibles))
      + ' combinaisons possibles à ce stade.')
print('Combinaison testée : ' + str(codebreaker((0,0))))
print()

init()
print('Il y a ' + str(len(combinaisons_possibles))
      + ' combinaisons possibles à ce stade.')
print('Combinaison testée : ' + str(codebreaker((0,0))))
print('Il y a ' + str(len(combinaisons_possibles))
      + ' combinaisons possibles à ce stade.')
print('Combinaison testée : ' + str(codebreaker((0,0))))
print('Il y a ' + str(len(combinaisons_possibles))
      + ' combinaisons possibles à ce stade.')
print('Combinaison testée : ' + str(codebreaker((0,0))))
print('Il y a ' + str(len(combinaisons_possibles))
      + ' combinaisons possibles à ce stade.')
print('Combinaison testée : ' + str(codebreaker((0,0))))
print('Il y a ' + str(len(combinaisons_possibles))
      + ' combinaisons possibles à ce stade.')
print('Combinaison testée : ' + str(codebreaker((0,0))))
print()
```

```
Il y a 4096 combinaisons possibles à ce stade.
Combinaison testée : GMMO
Il y a 4095 combinaisons possibles à ce stade.
Combinaison testée : MGBM
Il y a 4094 combinaisons possibles à ce stade.
Combinaison testée : VBVO
Il y a 4093 combinaisons possibles à ce stade.
Combinaison testée : VRRM
```

```
Il y a 4096 combinaisons possibles à ce stade.
Combinaison testée : GNVN
Il y a 4095 combinaisons possibles à ce stade.
Combinaison testée : MMBV
Il y a 4094 combinaisons possibles à ce stade.
Combinaison testée : BOVM
Il y a 4093 combinaisons possibles à ce stade.
Combinaison testée : NBMM
```

FIGURE 6 – Console du test du fichier codebreaker1

FIGURE 5 – Programme test du fichier codebreaker1

Ces tests nous permettent de voir que la fonction *init* réinitialise correctement les combinaisons non testées et que la fonction *codebreaker* renvoie bien une combinaison n'ayant pas déjà été testée et qu'elle la supprime des combinaisons non testées.

2.4 Fonction donner_possibles

Fonctionnement :

Nous commençons par récupérer toutes les possibilités grâce au produit d'ensemble d'itertools. Puis pour chaque combinaison de cet ensemble nous comparons l'évaluation de la combinaison récupérée en argument de la fonction et la combinaison de l'ensemble des possibilités, avec l'évaluation donnée en argument. Si ces deux évaluations sont identiques alors la combinaison de l'ensemble est encore possible donc nous l'ajoutons dans notre variable set. Dès que nous avons testé toutes les possibilités nous retournons la variable set créée avec toutes les combinaison encore possible.

Tests :

```
print('Possibilités après avoir essayé RRRR '
      + 'et reçu comme évaluation (4,0) : ')
print(donner_possibles('RRRR', (4,0)), '\n')

print('Possibilités après avoir essayé BBBB '
      + 'et reçu comme évaluation (3,0) : ')
print(donner_possibles('BBBB', (3,0)), '\n')

print('Possibilités après avoir essayé RVBJ '
      + 'et reçu comme évaluation (3,0) : ')
print(donner_possibles('RVBJ', (3,0)), '\n')
```

FIGURE 7 – Programme test de la fonction `donner_possibles`

Possibilités après avoir essayé RRRR et reçu comme évaluation (4,0) :
{'RRRR'}

Possibilités après avoir essayé BBBB et reçu comme évaluation (3,0) :
{'BBMB', 'BNBB', 'BBNB', 'BGBB', 'BBBB', 'BBBN', 'BBVB', 'MBBB', 'BBBO',
'BBJB', 'RBBB', 'BBBV', 'BBRB', 'BBBG', 'VBBB', 'OBBB', 'BVBB', 'BGBB',
'GBBB', 'BOBB', 'JBBS', 'BMBB', 'BRBB', 'BBOB', 'NBBB', 'BJBB', 'BBBJ',
'BBBR'}

Possibilités après avoir essayé RVBJ et reçu comme évaluation (3,0) :
{'OVBJ', 'RVVJ', 'RVBM', 'MVBj', 'RVMJ', 'GVBj', 'RVNJ', 'RVBB', 'RNBj',
'RVOJ', 'RVBR', 'RBBj', 'NVBJ', 'RVJJ', 'RGBj', 'ROBJ', 'JVBj', 'RVRj',
'RVBN', 'RVBG', 'RVGj', 'VBVj', 'VVBj', 'RJBj', 'RVBV', 'RVBO', 'RMBj',
'RRBJ'}

FIGURE 8 – Console du test de la fonction `donner_possibles`

Avec ces tests, nous voyons que la fonction `donner_possibles` retourne bien le bon set de possibilités.

2.5 Fonction `maj_possibles`

Fonctionnement :

Nous reprenons le même principe que `donner_possibles` sauf que nous limitons l'ensemble des possibles à celui pris en argument, pour éviter de vérifier des combinaisons que nous savons impossibles. Dès que nous avons récupéré l'ensemble des combinaisons encore possibles après l'essai et l'évaluation, nous mettons à jour l'ensemble pris en argument en lui faisant l'intersection avec l'ensemble des combinaisons possibles trouvé précédemment.

Tests :

```
test = donner_possibles('RVBJ', (3,0))
print('\nPossibilités après avoir essayé RVBJ '
      + 'et reçu comme évaluation (3,0) : ')
print(test)

maj_possibles(test, 'RVBJ', (3,0))
print('\nPossibilités après avoir essayé RVBJ '
      + 'et reçu comme évaluation (3,0) : ')
print(test)

maj_possibles(test, 'RVRV', (2,0))
print('\nPossibilités après avoir essayé RVRV '
      + 'et reçu comme évaluation (2,0) : ')
print(test)

maj_possibles(test, 'RVBM', (4,0))
print('\nPossibilités après avoir essayé RVBM '
      + 'et reçu comme évaluation (4,0) : ')
print(test)
```

FIGURE 9 – Programme test de la fonction `maj_possibles`

Possibilités après avoir essayé RVBJ et reçu comme évaluation (3,0) :
{'RBBj', 'RVGj', 'RJBj', 'RGBj', 'RVBR', 'NVBJ', 'RVMJ', 'OVBJ', 'MVBj',
'RVBM', 'BVBJ', 'RVBV', 'RVRj', 'RVBG', 'RVJJ', 'RMBj', 'RVBB', 'RVBN',
'RRBJ', 'ROBJ', 'RVVj', 'VVBj', 'RVNJ', 'RVBO', 'RVOj', 'RNBj', 'JVBj',
'GVBj'}

Possibilités après avoir essayé RVBJ et reçu comme évaluation (3,0) :
{'RBBj', 'RVGj', 'RJBj', 'RGBj', 'RVBR', 'NVBJ', 'RVMJ', 'OVBJ', 'MVBj',
'RVBM', 'BVBJ', 'RVBV', 'RVRj', 'RVBG', 'RVJJ', 'RMBj', 'RVBB', 'RVBN',
'RRBJ', 'ROBJ', 'RVVj', 'VVBj', 'RVNJ', 'RVBO', 'RVOj', 'RNBj', 'JVBj',
'GVBj'}

Possibilités après avoir essayé RVRV et reçu comme évaluation (2,0) :
{'RVMJ', 'RVOj', 'RVJJ', 'RVBB', 'RVGj', 'RVBM', 'RVBN', 'RVNJ', 'RVBG',
'RVBO'}

Possibilités après avoir essayé RVBM et reçu comme évaluation (4,0) :
{'RVBM'}

FIGURE 10 – Console du test de la fonction `maj_possibles`

On vérifie ici que la fonction nous réduit correctement le set des possibilités et que si on utilise deux fois la fonction avec la même combinaison, le set ne change pas.

2.6 Codebreaker2

Fonctionnement :

À chaque début de partie, nous créons l'ensemble de toutes les possibilités selon `LENGTH` et `COLORS` du fichier `common.py`. Le premier choix se fait aléatoirement puis nous utilisons

maj_possibles pour l'ensemble des possibilités restantes avec l'évaluation faite par le codemaker pour l'essai que nous avons fait au tour précédent. Puis nous choisissons aléatoirement une combinaison dans ce nouvel ensemble.

Tests :

```

init()
print('Combinaison testée initialement :', codebreaker(None))
init()
print('Combinaison testée initialement :', codebreaker(None))
init()
print('Combinaison testée initialement :', codebreaker(None), '\n\n')

init()
codebreaker(None)
print('On test', combinaison_testee, 'et on a pour évaluation (3,0)\n')

test = common.donner_possibles(combinaison_testee, (3,0))
codebreaker((3,0))
if test == combinaisons_possibles:
    print('Les combinaisons possibles sont correctement déterminées.')
print('On test', combinaison_testee, 'et on a pour évaluation (3,0)')
if combinaison_testee in combinaisons_possibles:
    print(combinaison_testee, 'était bien une combinaison possible.\n')

common.maj_possibles(test, combinaison_testee, (3,0))
codebreaker((3,0))
if test == combinaisons_possibles:
    print('Les combinaisons possibles sont correctement déterminées.')
print('On test', combinaison_testee, 'et on a pour évaluation (3,0)')
if combinaison_testee in combinaisons_possibles:
    print(combinaison_testee, 'était bien une combinaison possible.\n')

common.maj_possibles(test, combinaison_testee, (3,0))
codebreaker((3,0))
if test == combinaisons_possibles:
    print('Les combinaisons possibles sont correctement déterminées.')
print('On test', combinaison_testee, 'et on a pour évaluation (4,0)')
if combinaison_testee in combinaisons_possibles:
    print(combinaison_testee, 'était bien une combinaison possible.')

```

Combinaison testée initialement : NVVG
Combinaison testée initialement : OJMO
Combinaison testée initialement : OG00

On test JORM et on a pour évaluation (3,0)

Les combinaisons possibles sont correctement déterminées.
On test JJRM et on a pour évaluation (3,0)
JJRM était bien une combinaison possible.

Les combinaisons possibles sont correctement déterminées.
On test JNRM et on a pour évaluation (3,0)
JNRM était bien une combinaison possible.

Les combinaisons possibles sont correctement déterminées.
On test JM RM et on a pour évaluation (4,0)
JM RM était bien une combinaison possible.

FIGURE 12 – Console du test du fichier codebreaker2

FIGURE 11 – Programme test du fichier codebreaker2

On vérifie tout d'abord que la fonction codebreaker retourne une combinaison au hasard.

On teste ensuite que le codebreaker met correctement à jour sa mémoire des combinaisons possibles et qu'il sélectionne une combinaison possible.

2.7 Codemaker2

Fonctionnement :

Ce codemaker aura le droit à un peu de rouerie, c'est-à-dire qu'il peut modifier sa solution tant qu'elle respecte les évaluations précédentes. Nous allons donc choisir l'évaluation qui réduit le moins l'ensemble des possibilités, car nous n'avons pas besoin de choisir une solution en particulier. Pour cela, nous créons un dictionnaire qui prend en clé l'évaluation faite pour cette solution avec l'essai du codebreaker et en valeur le nombre de solutions ayant cette évaluation. Nous remplissons ce dictionnaire avec les combinaisons encore possible suivant les évaluations faites précédemment. On regarde ensuite l'évaluation qui a le plus grand nombre de solutions possibles (ce sera la taille de l'ensemble des combinaisons encore possibles après cette évaluation). Cette évaluation sera renvoyée au codebreaker et nous mettons à jour l'ensemble des combinaisons encore possibles suivant cette évaluation et les évaluations précédentes.

Tests :

```
init()
print('Solution initiale :', solution, '\n')
while(len(combinaisons_possibles) > 1):
    attempt = common.choices_v2(list(combinaisons_possibles))
    print('On suppose que le codebreaker essaie', attempt)
    temp = len(combinaisons_possibles)
    print('On donne', codemaker(attempt),
          'avec comme nouvelle solution', solution)
    print('On a conservé',
          int(len(combinaisons_possibles)/temp*10000)/100,
          '% des combinaisons précédentes.')
    print('Il reste exactement', len(combinaisons_possibles),
          'combinaisons possibles.\n')
```

FIGURE 13 – Programme test du fichier
codemaker2

Solution initiale : JORM

On suppose que le codebreaker essaie OOBN
On donne (0, 1) avec comme nouvelle solution VBJV
On a conservé 28.32 % des combinaisons précédentes.
Il reste exactement 1160 combinaisons possibles.

On suppose que le codebreaker essaie NJVV
On donne (0, 1) avec comme nouvelle solution JBRM
On a conservé 24.13 % des combinaisons précédentes.
Il reste exactement 280 combinaisons possibles.

On suppose que le codebreaker essaie GNRR
On donne (0, 1) avec comme nouvelle solution MRJO
On a conservé 28.57 % des combinaisons précédentes.
Il reste exactement 80 combinaisons possibles.

On suppose que le codebreaker essaie MGJO
On donne (0, 2) avec comme nouvelle solution RBMJ
On a conservé 18.75 % des combinaisons précédentes.
Il reste exactement 15 combinaisons possibles.

On suppose que le codebreaker essaie BRMJ
On donne (0, 2) avec comme nouvelle solution VBGM
On a conservé 33.33 % des combinaisons précédentes.
Il reste exactement 5 combinaisons possibles.

On suppose que le codebreaker essaie JBGB
On donne (2, 0) avec comme nouvelle solution VBGM
On a conservé 40.0 % des combinaisons précédentes.
Il reste exactement 2 combinaisons possibles.

On suppose que le codebreaker essaie VMGB
On donne (2, 2) avec comme nouvelle solution VBGM
On a conservé 50.0 % des combinaisons précédentes.
Il reste exactement 1 combinaisons possibles.

FIGURE 14 – Console du test du fichier
codemaker2

On peut vérifier que le codemaker prend une solution compatible avec les évaluations précédentes et qu'il donne l'évaluation correspondante. Nous montrons la solution choisie même si elle n'est pas nécessaire au codemaker comme dit précédemment.

On voit que le codemaker arrive à peu réduire les combinaisons possibles à chaque appel.

2.8 Codebreaker3

Fonctionnement :

L'objectif de ce codebreaker est de tester la combinaison optimale à chaque moment de la partie, qu'elle soit techniquement possible ou non. Nous regardons pour chaque combinaison que l'on pourrait tester ce qu'il se passerait dans le pire des cas, c'est-à-dire lorsque l'ensemble des solutions encore possibles après cet essai est le plus grand. Puis nous comparons, chaque longueur de l'ensemble des solutions encore possibles dans le pire cas pour chaque combinaison que nous n'avons toujours pas testé. On regroupe celles qui ont la taille de cet ensemble le plus petit et nous les enregistrons dans un set.

Nous regardons ensuite dans l'ensemble des combinaisons qui nous donnent la même quantité d'information, lesquelles appartiennent aux solutions encore possibles. Si cette intersection d'ensemble est non-vide alors nous testons une des combinaisons qui est possible tout en gardant la même quantité d'information. Sinon nous testons une combinaison qui est théoriquement impossible, mais qui nous donne plus d'information qu'une combinaison plausible.

Comme nous savons que nous allons lancer plusieurs partie d'affilée pour la réalisation des graphes, nous décidons d'enregistrer l'ensemble des combinaisons optimales du premier tour pour

la longueur et l'ensemble des couleurs défini dans un dictionnaire. Puis à tous les premiers tours, nous choisissons une combinaison aléatoirement dans cette liste de combinaisons optimales. Cela optimise notre programme lorsque nous faisons tourner beaucoup de parties, car il nous évite de faire des millions de tours de boucle inutiles.

Tests :

```
#On test à la main avec comme solution BJNM
import codebreaker3
play_human_against_codebreaker(codebreaker3)
```

FIGURE 15 – Programme test du fichier codebreaker3

```
Combinaisons de taille 4, couleurs disponibles
Combinaison proposée: NRVO

Saisir nombre de plots rouges: 0

Saisir nombre de plots blancs: 1
Essai 1 : NRVO (0,1)
Combinaison proposée: GMMG

Saisir nombre de plots rouges: 0

Saisir nombre de plots blancs: 1
Essai 2 : GMMG (0,1)
Combinaison proposée: MGBB

Saisir nombre de plots rouges: 0

Saisir nombre de plots blancs: 2
Essai 3 : MGBB (0,2)
Combinaison proposée: JBNM

Saisir nombre de plots rouges: 2

Saisir nombre de plots blancs: 2
Essai 4 : JBNM (2,2)
Combinaison proposée: BJNM

Saisir nombre de plots rouges: 4

Saisir nombre de plots blancs: 0
Essai 5 : BJNM (4,0)
Le codebreaker a trouvé BJNM en 5 essais
```

FIGURE 16 – Console du test du fichier codebreaker3

On observe que le codebreaker ne tente pas que des combinaisons possibles, et arrive à trouver correctement la solution en très peu de coups.

2.9 Fonction play_log

Fonctionnement :

On commence par ouvrir ou créer le fichier dont le nom est donné en argument et appeler les fonctions *init* du codebreaker et codemaker.

Puis on joue une partie en appelant dans une boucle infinie successivement le codebreaker et le codemaker et en écrivant dans le fichier ce qu'ils nous retournent. Lorsque l'évaluation de notre codemaker nous indique que le codebreaker a trouvé la solution, on sort de la boucle.

Tests :

```
import codebreaker2
import codemaker2
play_log(codemaker2, codebreaker2, 'test.txt')
```

FIGURE 17 – Programme test de la fonction
play_log

Fichier
OJJO
0,0
RRGV
0,1
NGBM
1,1
BVMM
1,1
MGMG
0,0
NVNB
1,2
VVBN
4,0

FIGURE 18 – Fichier du test de la fonction
play_log

On observe que le programme remplit le fichier comme attendu.

2.10 Check_codemaker

Fonctionnement :

On commence par ouvrir le fichier dont le nom nous est donné en argument et on enregistre dans les tableaux *combinaisons* et *evaluations* ses données.

On détermine la solution de la partie en regardant le dernier élément de *combinaisons* qui correspond forcément à la solution car une partie se finit quand le codebreaker la donne.

Il suffit alors de vérifier à l'aide d'une boucle que chaque évaluation donnée par le codemaker est en adéquation avec ce que l'on obtient en appelant *evaluation* sur les tentatives du codebreaker avec la solution finale.

Tests :

```
#Test où le fichier est issu d'une vraie partie
check_codemaker('test1.txt')

#Test où on change la 1ere évaluation
check_codemaker('test2.txt')

#Test où on change la 6e évaluation
check_codemaker('test3.txt')
```

FIGURE 19 – Programme test de la fonction
check_codemaker

Fichier	Fichier	Fichier
OJJO	OJJO	OJJO
0,0	2,0	0,0
RRGV	RRGV	RRGV
0,1	0,1	0,1
NGBM	NGBM	NGBM
1,1	1,1	1,1
BVMM	BVMM	BVMM
1,1	1,1	1,1
MGMG	MGMG	MGMG
0,0	0,0	0,0
NVNB	NVNB	NVNB
1,2	1,2	0,0
VVBN	VVBN	VVBN
4,0	4,0	4,0

FIGURE 20 – Fichiers du test de la fonction
check_codemaker

```

Il semblerait que le codemaker n'ait pas triché.
Le codemaker a visiblement triché !
Le codemaker a visiblement triché !

```

FIGURE 21 – Console du test de la fonction `check_codemaker`

On voit que le programme identifie correctement si le codemaker a visiblement triché ou non.

3 Questions

3.1 Question 3 - Analyse de la performance du codebreaker0

Notre programme `codebreaker0` envoie une combinaison aléatoire. Il envoie donc une des $\text{len}(\text{COLORS})^{\text{LENGTH}}$ (que l'on notera N par la suite) possibilité de combinaison. On pose p la probabilité de tirer la bonne combinaison à un tirage. Sachant que l'on fait un tirage avec remise, on a $p = \frac{1}{N}$.

On note la variable aléatoire X qui correspond au nombre de tentatives après avoir réussi à tirer la bonne combinaison. Réussir au n -ième essai revient à échouer aux $(n-1)$ -ièmes essais précédents et de réussir à cet essai en particulier, on a donc :

$$\mathbb{P}(X = n) = \prod_{k=1}^{n-1} (1 - p) \times p = (1 - p)^{n-1} p$$

La formule de l'espérance est :

$$\mathbb{E}(X) = \sum_{k=1}^{+\infty} x_k \mathbb{P}(X = x_k) = \sum_{n=1}^{+\infty} n \mathbb{P}(X = n) = \sum_{n=1}^{+\infty} np(1 - p)^{n-1}$$

Sachant que $p \in [0; 1[$, on a $(1 - p) \in] - 1; 1[$ donc :

$$\mathbb{P}(X = x_k) = p \times \sum_{n=1}^{+\infty} n(1 - p)^{n-1} = p \times \frac{1}{(1 - (1 - p))^2} = \frac{1}{p} = N$$

On a donc une espérance du nombre d'essai avant de trouver la bonne de combinaison correspondant à $N = \text{len}(\text{COLORS})^{\text{LENGTH}}$.

Maintenant que l'on a déterminé théoriquement l'espérance du nombre d'essais, voyons si notre programme donne le même résultat. Nous allons réaliser ce test pour $\text{len}(\text{COLORS}) = 8$ et $\text{LENGTH} = 4$, ce qui donne une espérance théorique de 4096. Pour cela nous allons afficher via le programme `histogramme`, la répartition du nombre d'essais de 10 000 parties, ainsi que calculer, grâce à la fonction `numpy.mean`, la moyenne du nombre d'essais pour ces 10 000 parties. Nous avons donc l'histogramme et le texte suivant :

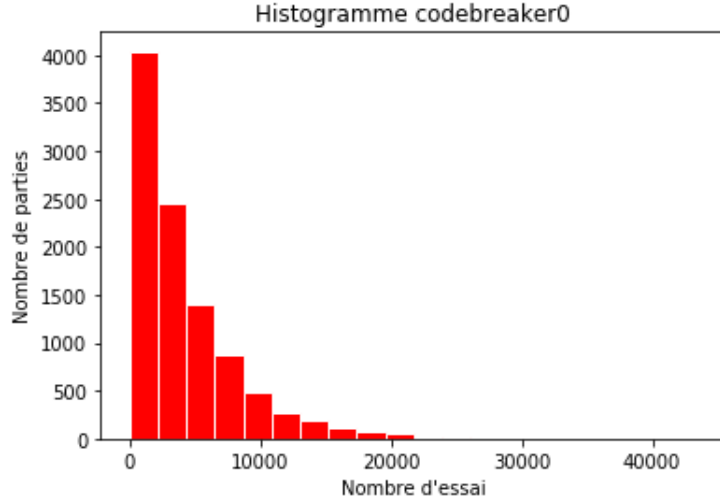


FIGURE 22 – Histogramme du codebreaker0 contre le codemaker1
La moyenne des valeurs de l'histogramme sur 10000 parties est 4123.6247

On remarque que les nombres de tentatives sont principalement concentrés vers des valeurs faibles malgré certaines tentatives qui atteignent des valeurs très importantes (jusqu'à 5 fois l'espérance).

Nous avons trouvé une espérance de 4123,6247, ce qui est proche de ce que nous avons déterminé par le calcul.

3.2 Question 4 - Analyse de la performance du codebreaker1

Le programme codebreaker1 comme est indiqué dans son explication envoie une combinaison qui n'a pas été testée, pas déjà envoyé. On pose p_k la probabilité de tirer la bonne combinaison à un k -ième tirage. Choisir une combinaison aléatoirement qui n'a pas encore été essayé revient à faire un tirage sans remise de combinaison. Sachant que l'on fait un tirage sans remise, on a $p = \frac{1}{\text{len}(\text{COLORS})^{\text{LENGTH}} - k}$. Pour simplifier l'écriture, on pose $N = \text{len}(\text{COLORS})^{\text{LENGTH}}$. On note la variable aléatoire X qui correspond au nombre de tentatives après avoir réussi à tirer la bonne combinaison. Cette fois-ci, on ne peut tirer au maximum que N fois, car on aura au bout des N essais, tiré toutes les possibilités et donc assurément la bonne. Réussir au n -ième essai revient à échouer aux $(n-1)$ -ièmes essais précédents et de réussir à cet essai en particulier, on a donc :

$$\begin{aligned} \mathbb{P}(X = n) &= \prod_{k=1}^{n-1} (1 - p_k) \times p_n = \left(1 - \frac{1}{N}\right) \left(1 - \frac{1}{N-1}\right) \dots \left(1 - \frac{1}{N-(n-1)}\right) \frac{1}{N-n} \\ &= \frac{N-1}{N} \frac{N-2}{N-1} \dots \frac{N-n}{N-1-n} \frac{1}{N-n} = \frac{1}{N} \end{aligned}$$

La formule de l'espérance est :

$$\mathbb{E}(X) = \sum_{k=1}^{+\infty} x_k \mathbb{P}(X = x_k) = \sum_{n=1}^N n \mathbb{P}(X = n) = \sum_{n=1}^N \frac{n}{N} = \frac{1}{N} \frac{N(N+1)}{2} = \frac{N+1}{2}$$

On a donc une espérance du nombre d'essai avant de trouver la bonne de combinaison correspondant à $N = \frac{N+1}{2} = \frac{\text{len}(\text{COLORS})^{\text{LENGTH}} + 1}{2}$. On peut donc s'attendre à diviser le nombre d'essais avant réussite par 2.

Nous allons maintenant voir si le programme est bel est bien aussi efficace. Le test que nous avons réalisé est fait avec $\text{len}(\text{COLORS}) = 8$ et $\text{LENGTH} = 4$, on a donc une espérance théorique de 2048,5. Pour observer le gain d'efficacité, nous avons tracé l'histogramme de 10 000 parties réalisées avec le `codemaker1` le `codebreaker1`.

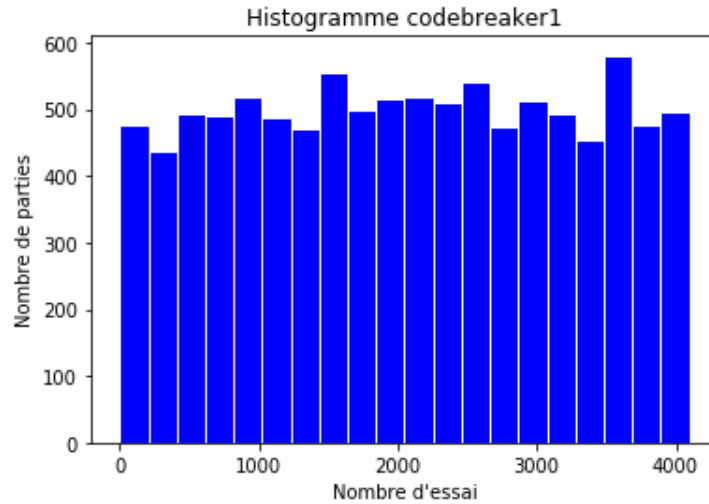


FIGURE 23 – Histogrammes du codebreaker1 contre le codemaker1

La moyenne des valeurs de l'histogramme pour le codebreaker1 sur 10000 parties est 2066.7293

On a aussi calculé la moyenne des tentatives, 2066,7293, ce qui est très proche de l'espérance théorique calculé et qui correspond presque à diviser la moyenne de la question précédente par 2. Pour mieux comparer les deux fonctions, nous pouvons mettre ensemble les deux histogrammes ensemble (celui du `codebreaker0` et du `codebreaker1`) sur un seul graphique. Cependant, nous avons aussi choisi d'afficher un violin plot pour que cela plus lisible.

Le violin plot permet de voir, comme on l'a paramétré, de voir la plage de répartition du nombre de tentative ainsi que la moyenne. Le violin plot est un graphique proche de ce qu'on appelle la boîte à moustaches, mais n'affichant pas la valeur médiane et les quartiles obligatoirement et montrant plus la densité avec des formes arrondies. Le violin plot de matplotlib permet contrairement au graphe blox plot de choisir ce que l'on affiche et ne considère pas des points comme aberrants et qu'il mettrait à part.

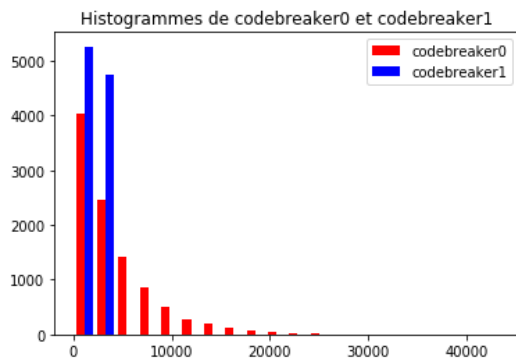


FIGURE 24 – Histogrammes du codebreaker0 contre le codemaker1 et du codebreaker1 contre le codemaker1

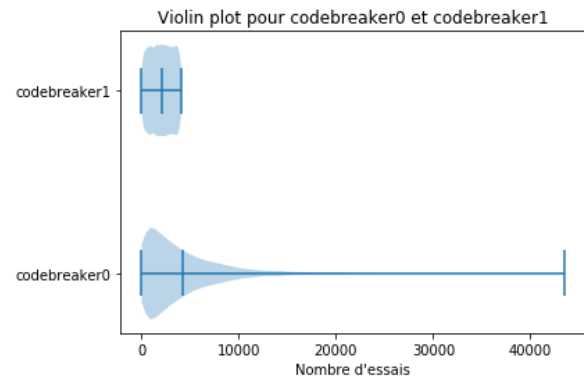


FIGURE 25 – Violin plot du codebreaker0 contre le codemaker1 et du codebreaker1 contre le codemaker1

On voit donc sur le violin plot que la moyenne du codebreaker1 est plus faible que celle du codebreaker0 (trait au milieu de chaque figure). De plus, on voit bien que la plage pour le codebreaker1 est plus faible et est bien majoré par N . On remarque aussi que les nombres de tentatives pour codebreaker1 sont réparties de manière plus équitables sur la plage, ce qui fait que la moyenne se trouve au milieu de cette dernière. Alors que pour le codebreaker0, on a une répartition inégales avec la moyenne qui est au quart de la plage. Le violin plot du codebreaker0 nous montre aussi que la partie peut aller jusqu'à plus de 42 000 essais, ce qui est énorme (10 fois l'espérance!).

3.3 Question 7 - Analyse de la performance du codebreaker2

Pour cette question, nous devons implémenter le programme codebreaker2 que nous avons décrit dans une partie précédente.

Intéressons-nous maintenant à l'efficacité de ce programme par rapport au codebreaker1. Pour cela, nous allons utiliser le codemaker1. Nous avons donc, avec la fonction associée à cette question dans le programme histogramme, l'histogramme suivant pour codebreaker2 :

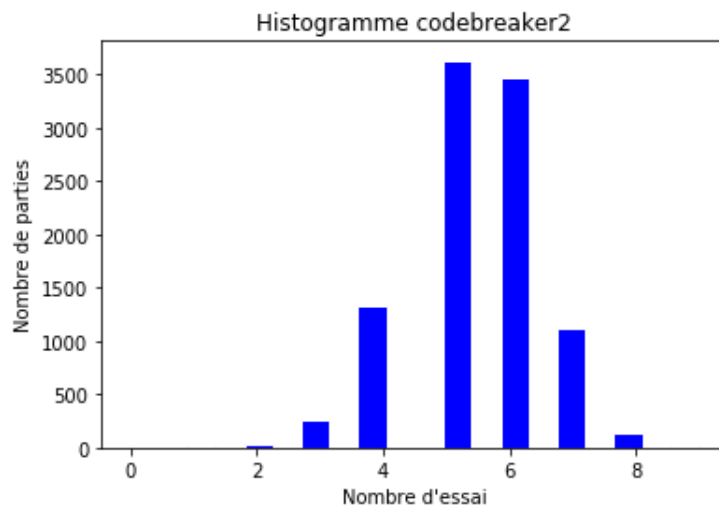


FIGURE 26 – Histogrammes du codebreaker2 contre le codemaker1

On remarque directement que le nombre de tentatives est drastiquement plus faible pour le codebreaker2. En effet, nous sommes au maximum à 8 tentatives pour le codebreaker2 tandis que pour le codebreaker1, nous sommes sur des milliers de tentatives. La différence se remarque lorsque l'on met cette histogramme et l'histogramme du codebreaker1 sur le même graphe :

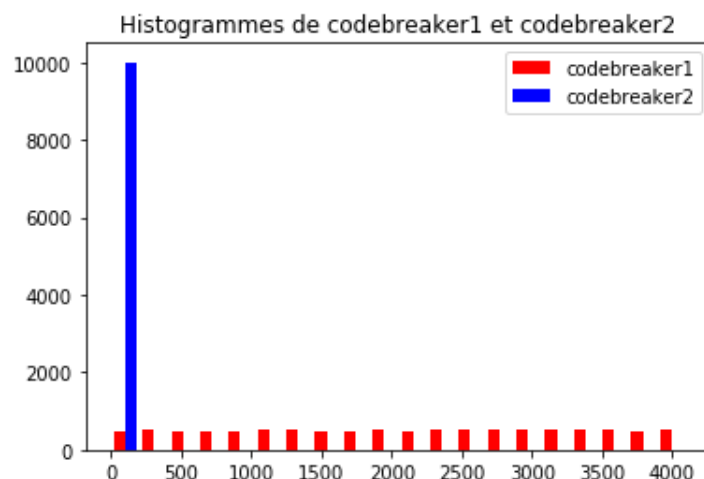


FIGURE 27 – Histogrammes du codebreaker1 contre le codemaker1 et du codebreaker2 contre le codemaker1

On a toutes les tentatives du codebreaker2 concentrées en une seule bande, tandis que le codebreaker1 s'étale entre 1 et 4096 (car les histogrammes ont été réalisés avec le COLORS et le LENGTH donné à la base).

Si on trace le violin plot des deux codebreakers, on obtient la figure suivante et remarquons que la plage du codebreaker2 est vraiment très petites par rapport à celle du codebreaker1.

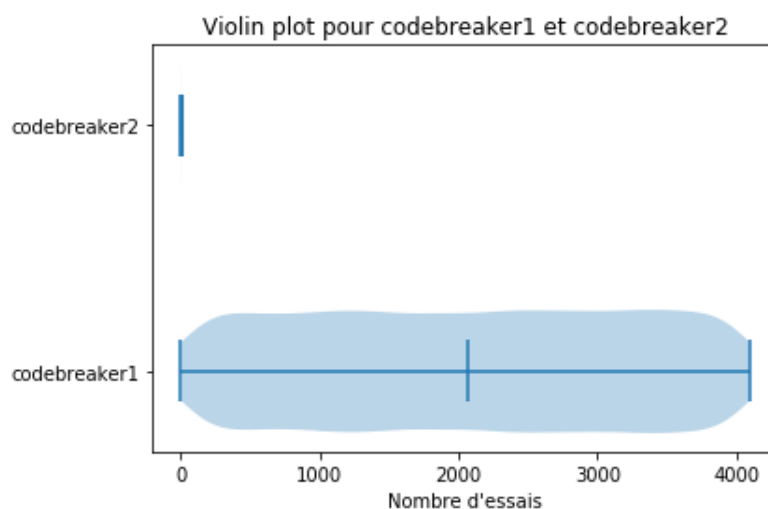


FIGURE 28 – Violin plot du codebreaker1 contre le codemaker1 et du codebreaker2 contre le codemaker1

Regardons maintenant les moyennes :

La moyenne des valeurs de l'histogramme pour le codebreaker2 sur 10000 parties est 5.4176

Cette différence se remarque bien avec le calcul des moyennes : nous avons 5,4176 pour le codebreaker2 et 2048 pour le codebreaker1, on a bien $5,4 \lll 2067$.

3.4 Question 8 - Analyse de la performance du codemaker2

On s'intéresse dans cette question à améliorer l'efficacité du codemaker en le faisant "tricher". Il faut donc cette fois-ci voir si le nombre de tentatives a augmenté entre le codemaker1 et le

codemaker2. On réalise donc X tests sur avec ces deux codemakers contre le codebreaker2. On obtient les histogrammes suivantes :

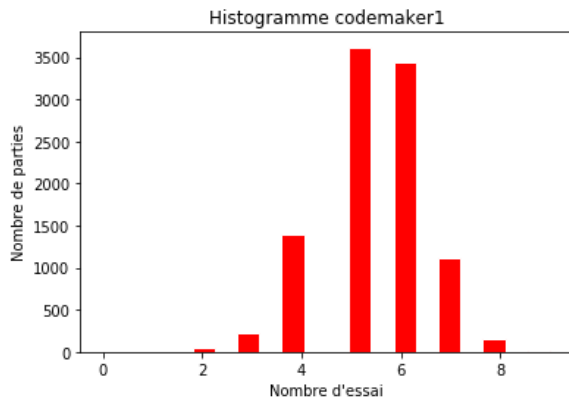


FIGURE 29 – Histogramme du codebreaker2 contre le codemaker1

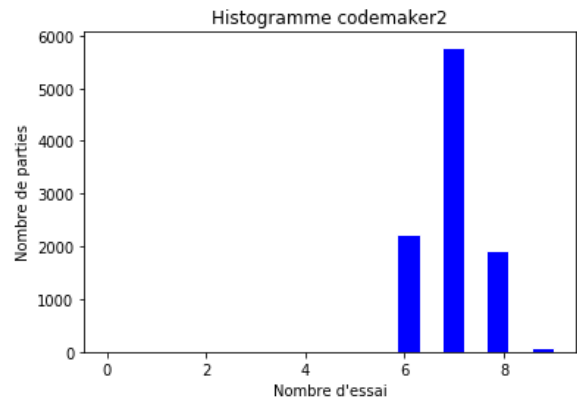


FIGURE 30 – Histogramme du codebreaker2 contre le codemaker2

Nous avons aussi les moyennes suivantes :

La moyenne des valeurs de l'histogramme pour le codemaker1 sur 10000 parties est 5.4174

La moyenne des valeurs de l'histogramme pour le codemaker2 sur 10000 parties est 6.9797

On remarque d'abord que la moyenne pour le codemaker2 est légèrement plus importante que la moyenne pour le codemaker1, ce qui est un premier signe d'efficacité pour le codemaker2.

Pour mieux comparer les deux histogrammes, met-on les ensemble sur un même graphe :

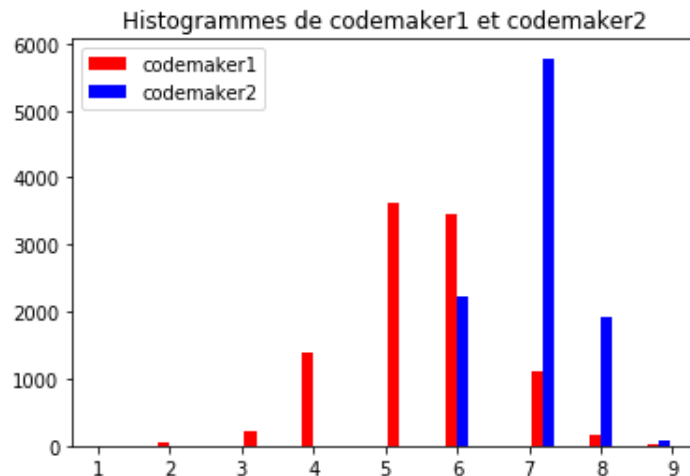


FIGURE 31 – Histogrammes du codebreaker2 contre le codemaker1 et du codebreaker2 contre le codemaker2

On remarque que les nombres de tentatives du codemaker2 sont généralement plus importantes que celle du codemaker1.

Regardons maintenant ce que donne le violin plot :

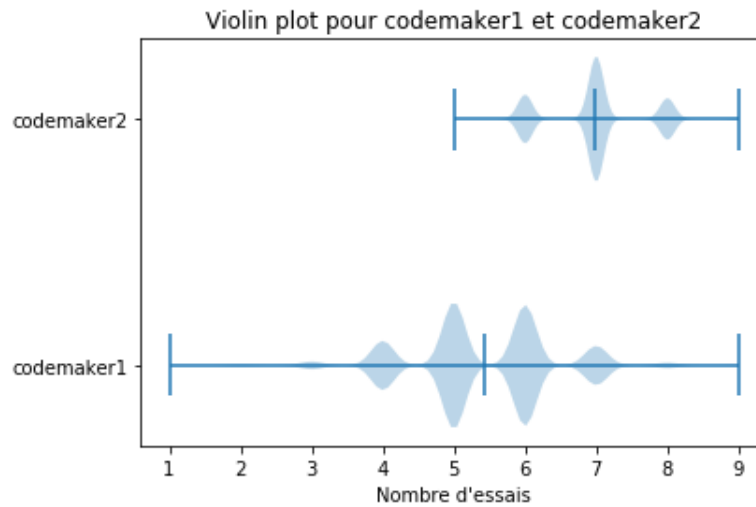


FIGURE 32 – Violin plot du codebreaker2 contre le codemaker1 et du codebreaker2 contre le codemaker2

On remarque bien que les valeurs pour le codemaker2 sont plus élevés que celles du codemaker1. Par contre, on remarque que les deux admettent un même maximum (9 tentatives). On en déduit donc que le codemaker2 n'augmente pas énormément le nombre de tentatives mais les concentrent vers des valeurs plus grandes. Le codemaker2 permet juste d'être sûr d'être dans la partie élevée de la plage du codemaker1.

3.5 Question 9 - Utilité de tester une combinaison impossible

Il est possible que tester une combinaison que l'on sait impossible apporte plus d'information que n'importe quelle solution possible.

Illustrons ceci avec un exemple :

On se place dans une situation où l'on a 8 couleurs : A, B, C, D, E, F, G et H, et que chaque combinaisons est faite de 4 plots. Imaginons les coups/évaluation suivants :

AAAA : (3,0)

AAAB : (3,0)

Nous sommes donc désormais certains que la solution commence par AAA.

En suivant la logique de ne tester que des combinaisons possibles, dans le pire des cas, ou avec un codemaker modifiant sa solution, on se retrouve à effectuer la suite de coups/évaluations suivantes :

AAAC : (3,0)

AAAD : (3,0)

AAAE : (3,0)

AAAF : (3,0)

AAAG : (3,0)

AAAH : (4,0)

On a alors dans ce cas une partie en 8 coups.

Maintenant, reprenons les mêmes premiers coups, et essayons une stratégie différente.

CDEF : (0,0)

AAAG : (3,0)

AAAH : (4,0)

Ce qui nous fait cette fois ci un total de 5 coups dans le pire des cas.

PS : Si nous avions eu pour CDEF : (0,1), nous aurions alors su que les 3 solutions possibles étaient AAAC, AAAD et AAAE et il aurait suffi de tester AACD pour savoir laquelle des 3 était la bonne. En effet (2,0) nous montrait qu'il s'agissait de AAAE, (3,0) pour AAAD et (2,1) pour AAAC. Cela nous aurait alors également fait un total de 5 coups.

Si nous avions eu CDEF : (1,0), alors AAAG était la seule solution possible et donc notre partie se finissait en 4 coups.

On a donc bien ici un exemple qui nous prouve que dans certains cas tester que des combinaisons possibles n'est pas la meilleure stratégie.

3.6 Question 10 - Analyse de la performance du codebreaker3

Nous nous intéressons dans cette question à l'application du programme codebreaker3. Ce programme doit permettre de réduire encore plus le nombre d'essais que le programme codebreaker2. Nous avons vu dans la question précédente qu'il est intéressant dans certains cas que nous testions une combinaison que l'on sait qu'elle n'est pas solution, qu'elle est impossible. Nous avons donc intégré ce raisonnement dans le programme codebreaker3. Regardons si ce programme apporte bel et bien un gain intéressant pour le nombre d'essais avant de trouver la solution. Nous avons donc réalisé dans le programme histogramme une fonction pour cette question faisant 1000 différentes parties avec le codebreaker2 et le codebreaker3 contre le codemaker1 et le codemaker2 pour observer si nous avons un gain entre le codebreaker2 et le codebreaker3. Voici les histogrammes contre le codemaker1 ainsi que les moyennes.

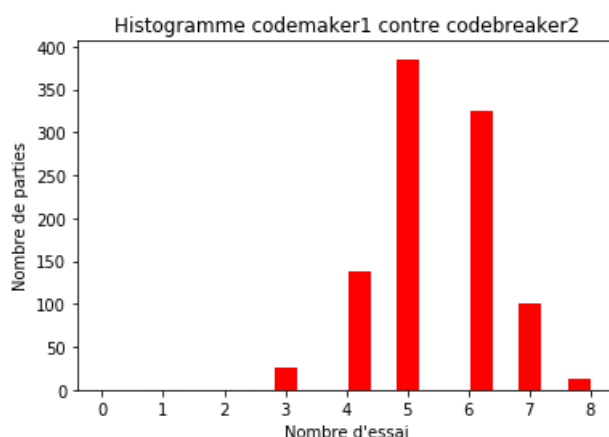


FIGURE 33 – Histogramme du codebreaker2 contre le codemaker1

La moyenne des valeurs de l'histogramme pour le codemaker1 contre codebreaker2 sur 1000 parties est 5.374

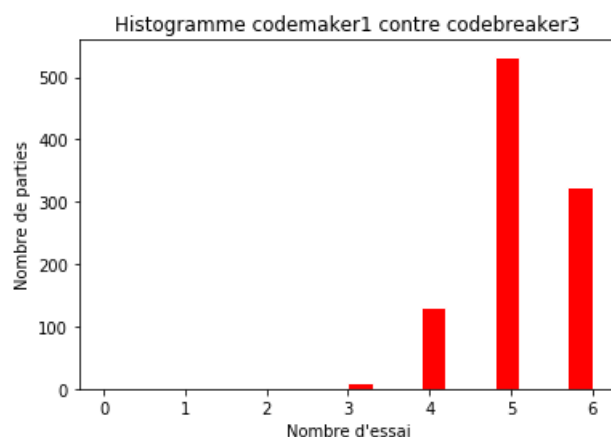


FIGURE 34 – Histogramme du codebreaker3 contre le codemaker1

La moyenne des valeurs de l'histogramme pour le codemaker1 contre codebreaker3 sur 1000 parties est 5.173

Voici les histogrammes des parties contre le codemaker2 et les moyennes.

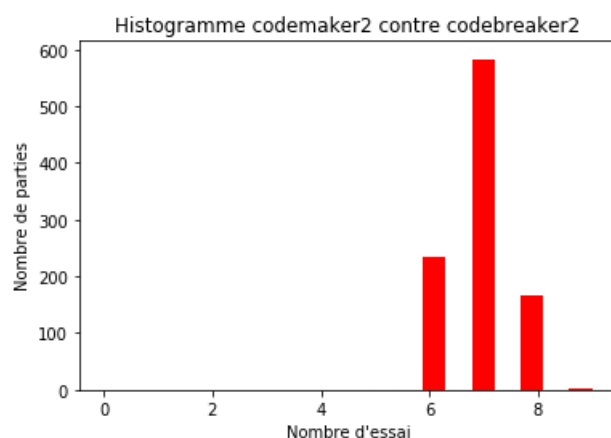


FIGURE 35 – Histogramme du codebreaker2 contre le codemaker2

La moyenne des valeurs de l'histogramme pour le codemaker2 contre codebreaker2 sur 1000 parties est 6.937

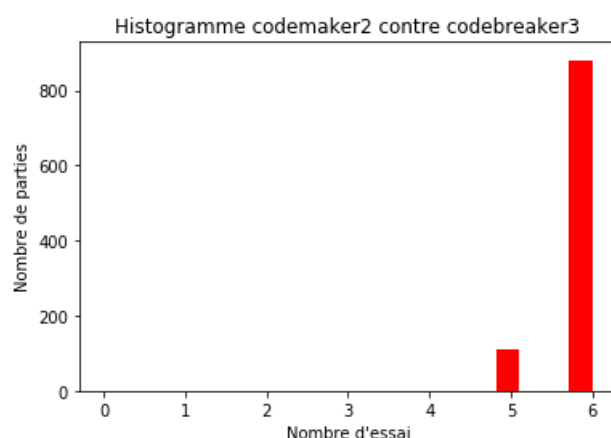


FIGURE 36 – Histogramme du codebreaker3 contre le codemaker2

La moyenne des valeurs de l'histogramme pour le codemaker2 contre codebreaker3 sur 1000 parties est 5.885

On remarque déjà que les moyennes pour le codebreaker3 sont inférieures aux moyennes pour le codebreaker2. De plus le codebreaker3 semble plus régulier pour le nombre d'essais ce qui déjà

un premier signe de gain.

Pour comparer au mieux les deux codebreaker2, regroupons les histogrammes et réalisons des violins plot :

Histogrammes de codebreaker2 et codebreaker3 contre codemaker1

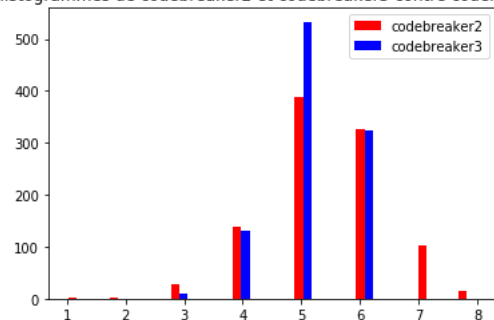


FIGURE 37 – Histogrammes du codemaker1 contre le codebreaker2 et le codebreaker3

Histogrammes de codebreaker2 et codebreaker3 contre codemaker2

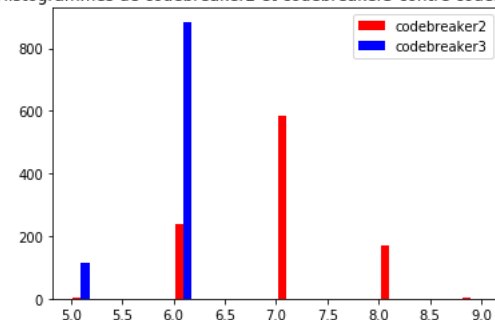


FIGURE 38 – Histogrammes du codemaker2 contre le codebreaker2 et le codebreaker3

Violin plot pour codebreaker2 et codebreaker3 contre codemaker1

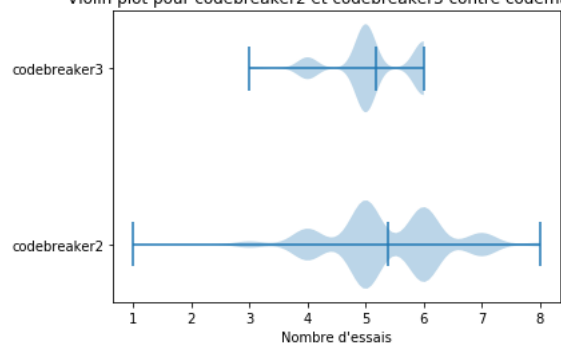


FIGURE 39 – Violin plot du codemaker1 contre le codebreaker2 et le codebreaker3

Violin plot pour codebreaker2 et codebreaker3 contre codemaker2

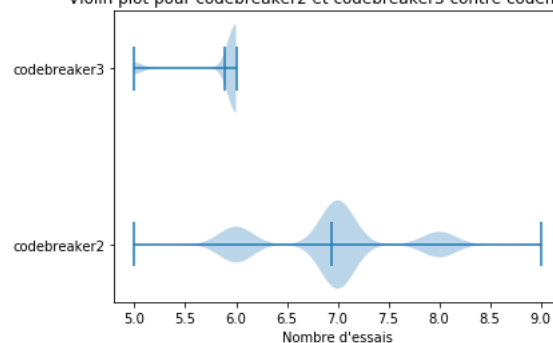


FIGURE 40 – Violin plot du codemaker2 contre le codebreaker2 et le codebreaker3

On peut observer que les plages de répartition pour le codebreaker3 sont plus petites que celle du codebreaker2. Cependant, elles sont plus petites en taille, mais sont pour des nombres d'essais plus faibles. On remarque bien aussi sur les violins plot que qu'importe les codemaker, le codebreaker3 arrive à finir une partie en 6 essais maximum, ce qui est un gain très intéressant du fait que le codebreaker2 semble ne finir une partie en 9 coups maximum contre le codemaker2 : cela représente un gain de 3 coups. Cependant, on peut remarquer que le codebreaker2 peut lui aussi réaliser les mêmes nombres d'essais que le codebreaker3. On peut alors noter sur les violins plot et les histogrammes que la majorité des nombres de tentatives est inférieur pour le codebreaker3 que pour le codebreaker2.

Le minimum de nombres d'essais n'est pas intéressant à comparer, ou difficile à juger, car chaque partie n'a pas la même solution et il est possible que par chance, l'aléatoire face que le codebreaker tombe par hasard sur la solution.

4 Conclusions - Question 13

Nous n'avons pas vraiment fait attention au temps exact que nous avons passé sur le projet mais on peut dire que l'on a passé environ 10 - 15 heures par personnes.

Concernant les difficultés, nous n'avons pas eu de problèmes particulier à répondre aux questions et à effectuer les implémentations demandées. Cependant rendre nos codes efficaces pour ne pas à avoir à attendre des heures pour leur exécution a été beaucoup plus ardu, en particulier pour le *codemaker2*. Nous avons commencer une première version où une partie contre le *codebreaker2* prenait une heure pour arriver à une version ou une partie dure une demi seconde dans le pire cas.

Nous avons chacun avancé en parallèle de notre côté, en s'entraidant lorsqu'une personne était en difficulté. Cette entraide a donc été un gain de temps. Nous avons fait le choix d'uploader nos codes sur Github et de faire notre compte rendu en Latex en ligne sur Overleaf pour pouvoir tous faire des modifications facilement.

Le travail de groupe a été intéressant car il a permis de voir comment coder les autres membres, de partager des astuces et de s'adapter à et comprendre d'autres manière de raisonner.

Ce travail nous a permis d'appliquer des notions et nouvelles variables vu en cours sur un cas vraiment concrets. Il nous a aussi permis de découvrir de nouvelles fonctions implémentées dans python en particulier l'application de dictionnaire, de set et la découverte d'*itertools* et de certaines fonctions de *matplotlib*.

Pour finir, nous pensons avoir rendu notre code le plus rapide possible même si il doit y avoir d'autres astuces pour gagner du temps sur le *codebreaker3*. De plus, après des milliers de parties jouées, en particulier pour faire les graphiques, nous n'avons pas remarquer de bugs évidents ou problématiques dans nos dernières versions des programmes.