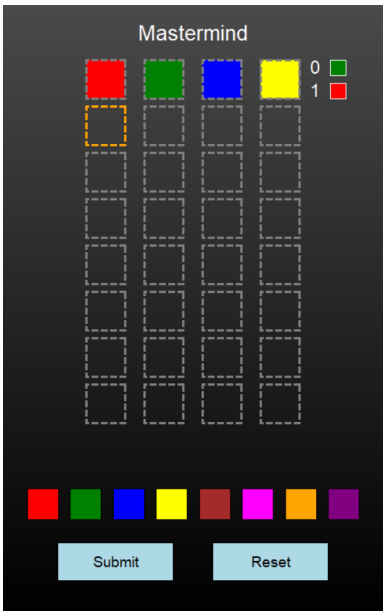
**La Prépa des INP - Grenoble**

Tronc commun - Informatique




# Mastermind

## Compte rendu de projet

**Auteurs :**
Joan Chassagne & Sofiène Faure–Claireau

**Professeur :**
Lionel

**Date de remise :** 25 avril 2025

# Table des matières

# 1 Introduction

## 1.1 Objectifs du TP

Dans le cadre de ce projet, nous avons implémenté en Python le jeu *Mastermind*, dans lequel un joueur (le codebreaker) doit deviner une combinaison secrète choisie par un autre joueur (le codemaker).

On a développé différentes versions de plus en plus performantes du *codebreaker*, en commençant par de simples essais aléatoires, jusqu'à des stratégies plus avancées. En parallèle, on a également amélioré le *codemaker*, avec notamment la possibilité de changer la combinaison à deviner pendant la partie afin de rallonger le jeu.

Les différentes versions du jeu ont été testées et analysées grâce à des graphiques, en se basant sur la rapidité du *codebreaker* à trouver la solution.

Enfin, on a réalisé une interface graphique pour rendre les parties plus interactives et amusantes.

## 1.2 Remarques

Pour chacune des questions, nous avons fournit une explication générale du code, nos tests ainsi que leur analyse. Vous trouverez les fichiers de code commentés en pièces jointes, ou bien sur ce dépôt Github.

Pour certaines questions, vous trouverez parfois plusieurs graphiques qui se ressemblent qui comparent un même *codebreaker* et un même *codemaker*. C'est simplement que notre fonction `histogramme(...args)` a été développée dynamiquement, et pour certains questions, il est plus intéressant visuellement d'avoir certains intervalles de regroupement de données (bins) plutôt que d'autres. Regardez donc simplement le graphique qui vous semble le mieux afficher les données.

## 1.3 Définition des termes

Dans cette section, nous définissons les termes utilisés tout au long de ce compte rendu ainsi que dans les commentaires du code.

- **Combinaison** : Suite de couleurs choisie par le *codemaker*, que le *codebreaker* doit deviner.
    - **Plots** : Eléments de la combinaison (une couleur représente un plot).
    - **Taille** (*LENGTH*) : Nombre de *plots* de la combinaison.
    - **Couleurs** (*COLORS*) : Couleur possibles des plots (plusieurs plots peuvent avoir la même couleurs).
- **Solution à deviner** : C'est la combinaison secrète que le *codebreaker* cherche à trouver.
- **Tentative** : Proposition faite par le *codebreaker* pour deviner la combinaison secrète.
- **Evaluation** : Réponse donnée par le *codemaker* après chaque tentative, indiquant

combien d'éléments sont corrects et bien placés, et combien sont corrects mais mal placés.

- **Codemaker** : Joueur qui choisit la combinaison secrète et évalue chacune des tentatives.
- **Codebreaker** : Joueur qui tente de deviner la combinaison secrète *codemaker*.
- **Tour de jeu** : Une tentative du *codebreaker* suivie d'une évaluation du *codemaker*.
- **Stratégie** : Méthode employée par le *codebreaker* pour optimiser ses tentatives et deviner la combinaison rapidement, et par le *codemaker* pour complexifier la tâche en modifiant la solution.

# 2 Questions

## 2.1 Question 1

Objectif

Écrire une fonction `évaluation(combinaison, combinaison référence)` dans *common.py* qui compare deux combinaisons et renvoie un couple d'entiers : le nombre de plots bien placés et le nombre de plots mal placés.

Explications du code

1. **Vérification des paramètres** :

   On s'assure que les deux fonctions sont de même longueur. En cas d'erreur, le programme affiche un message d'erreur et s'arrête.

2. **Mémoization** :

   Chaque fois qu'une évaluation est calculée, elle est enregistrée pour ne être recalculée. Si l'évaluation demandée a déjà été calculée, on la renvoie donc directement.

3. **Identification des plots bien placés** :

   Chaque plot de la combinaison est comparé à celui de la solution à la même position. S'il est bien placé, il est comptabilisé ; sinon il est ajouté à une liste pour vérifier s'il est seulement mal placé. On enregistre également la valeur correct attendue.

4. **Identification des plots mal placés** :

   Chaque plot mal placé est ensuite comparé aux plots de la solution qui n'ont pas encore été trouvés. S'il y est présent, il est comptabilisé et retiré pour éviter qu'il soit comptabilisé deux fois ; sinon on ne fait rien.

   *Par exemple, avec :*

   - `A=['R', 'G', 'B']` *: Les plots corrects non trouvés.*
   - `B=['G', 'G', 'M']` *: Les plots de la combinaison testée.*

   *Le premier plot* $G \in B$ *est dans* $A$*, donc on ajoute 1 au nombre de plots mal placé.*
   *Le deuxième plot* $G \in B$ *ne doit pas être compté à nouveau.*

5. **Retour des résultats** :

   On renvoie un couple contenant le nombre de plots bien placés, et le nombre de plots présents mais mal placés.

Tests & Analyses

Nous avons développé une fonction pour tester `evualation(...args)` . Tous les tests ont réussi.

```
1  def test_evaluation():
2
3    tests = [
4      # (combinaison, solution, resultat attendu)
5      ("RGBY", "RGB", "Erreur : les deux combinaisons n'ont pas la
           meme longueur"),  # "Erreur
6      ("RGBY", "RGBY", (4, 0)),  # Tout est bien place
7      ("RGBY", "YBGR", (0, 4)),  # Tout est mal place
8      ("RGBY", "RBGY", (2, 2)),  # Deux bien places, deux mal places
9      ("RRBB", "BBRR", (0, 4)),  # Deux couleurs inversees
10     ("RRGG", "RRBB", (2, 0)),  # Deux bien places, pas de mal
```

```
        places
    ("RRGG", "GGRR", (0, 4)),  # Tout mal place
    ("AAAA", "BBBB", (0, 0)),  # Aucune correspondance
]

for combinaison, solution, resultat_attendu in tests:
    resultat = evaluation(combinaison, solution)
    assert resultat == resultat_attendu, f"Echec : {combinaison} -
        {solution}, attendu {resultat_attendu}, obtenu {resultat}"

print("Tous les tests ont reussi !")
```

## 2.2 Question 2

Objectif

Ecrire une fonction `codemaker(combinaison)` dans *codemaker1.py*, capable d'évaluer les combinaisons testées en tenant compte des plots bien placés, et ceux présents, mais mal placés.

Explications du code

1. **Initialisation du *codemaker* :**

   A chaque début de partie, le *codemaker* génère aléatoirement et enregistre dans une variable globale, la solution.
2. **Evaluation :**

   On évalue la combinaison testée par rapport à la solution.
3. **Retour des résultats :**

   On renvoie cette évaluation.

Tests & Analyses

Aucun test n'est requis pour cette question. La fonction est assez claire, et, étant donné qu'elle fait directement appel à la fonction `evaluation(...args)` définie précédemment, cela reviendrait à répéter les mêmes tests.

## 2.3 Question 3

Objectif

Estimer l'espérance du nombre du nombre d'essaies nécessaires pour que la version 1 du *codebreaker* trouve la bonne solution.

Explications

**Essaie d'une combinaison :**

Le *codebreaker* essaie une combinaison.
Notons $p$ la probabilité d'essayer la bonne combinaison.

Notons $Y$ la variable aléatoire réelle qui correspond au numéro de la combinaison testée.
$Y$ suit une loi uniforme car :

- $Y(\Omega) = \left\{1, \ldots, \text{COLORS}^{\text{LENGTH}}\right\}$
- $\forall k \in \mathbb{N}^*, \mathbb{P}(Y = k) = \frac{1}{\text{COLORS}^{\text{LENGTH}}}$

Notons $X$ la variable aléatoire réelle telle que $X = 1$ si *codebreaker* a trouvé la bonne combinaison, $X = 0$ sinon. $X$ suit une loi de Bernoulli de paramètre $p$ car :

- $X(\Omega) = \{0, 1\}$
- $\mathbb{P}(X = 1) = p = \frac{1}{\text{COLORS}^{\text{LENGTH}}}$

**Essaies jusqu'à trouver la solution :**

On répète à l'infini et de façons indépendantes la même expérience de Bernoulli de paramètre $p$ et on attend le premier "succès".

Notons $N$ la variable aléatoire réelle qui compte le nombre de tentatives faites pour trouver la bonne combinaison.

Soit $n \in \mathbb{N}^*$

$$\begin{aligned}\{N = k\} &= \underbrace{\{X = 0\} \cap \cdots \cap \{X = 0\}}_{k-1 \text{ fois}} \cap \{X = 1\} \\ &= \bigcap_{i=1}^{n-1} \{X = 0\} \cap \{X = 1\}\end{aligned}$$

$$\begin{aligned}\mathbb{P}(N = k) &= P(X = 0)^{k-1} \times P(X = 1) \quad \text{Evènements 2 à 2 indépendants} \\ &= (1 - p)^{k-1} \times p\end{aligned}$$

$N$ suit une loi géométrique de paramètre $p$ car :

- $X(\Omega) = \mathbb{N}^*$
- $\forall k \in \mathbb{N}^*, \mathbb{P}(N = k) = p(1 - p)^{k-1}$

**Calcul de l'espérance :**

On sait que $\forall x \in ]-1, 1[, \sum_{n=1}^{+\infty} nx^{n-1} = \frac{1}{(1-x)^2}$

$$\begin{aligned}E(N) &= \sum_{k=1}^{+\infty} |k\mathbb{P}(N = k)| \\ &= \sum_{k=1}^{+\infty} \left[k \times \frac{1}{\text{COLORS}^{\text{LENGTH}}} \left(1 - \frac{1}{\text{COLORS}^{\text{LENGTH}}}\right)^{k-1}\right] \\ &= \frac{1}{\text{COLORS}^{\text{LENGTH}}} \sum_{k=1}^{+\infty} \left[k \times \left(1 - \frac{1}{\text{COLORS}^{\text{LENGTH}}}\right)^{k-1}\right] \\ &= \frac{1}{\text{COLORS}^{\text{LENGTH}}} \times \frac{1}{\left[1 - \left(1 - \frac{1}{\text{COLORS}^{\text{LENGTH}}}\right)\right]^2} \\ &= \text{COLORS}^{\text{LENGTH}} \\ &= 8^4 \\ &= 4096\end{aligned}$$

Cela signifie que pour un très grand nombre de partie jouées, on a en moyenne trouvé la solution au bout de 4096 tentatives.

Afin de vérifier notre approche théorique, nous avons simulé un grand nombre de parties (100 000), et enregistré le nombre d'essaies nécessaires au *codebreaker* pour trouver la solution.

Tests & Analyses

Le graphique montre une diminution rapide du nombre d'essaies nécessaires, confirmant que $N$ suit bien une loi géométrique. L'espérance observée, avoisinant 3840, reste proche de la valeur théorique de 4096, ce qui valide notre modèle analytique.

Cependant, ce modèle n'est pas représentatif de la réalité puisqu'on n'a pas autant de tentatives. Dans la suite, nous verrons d'autres stratégies pour réduire cette espérance.

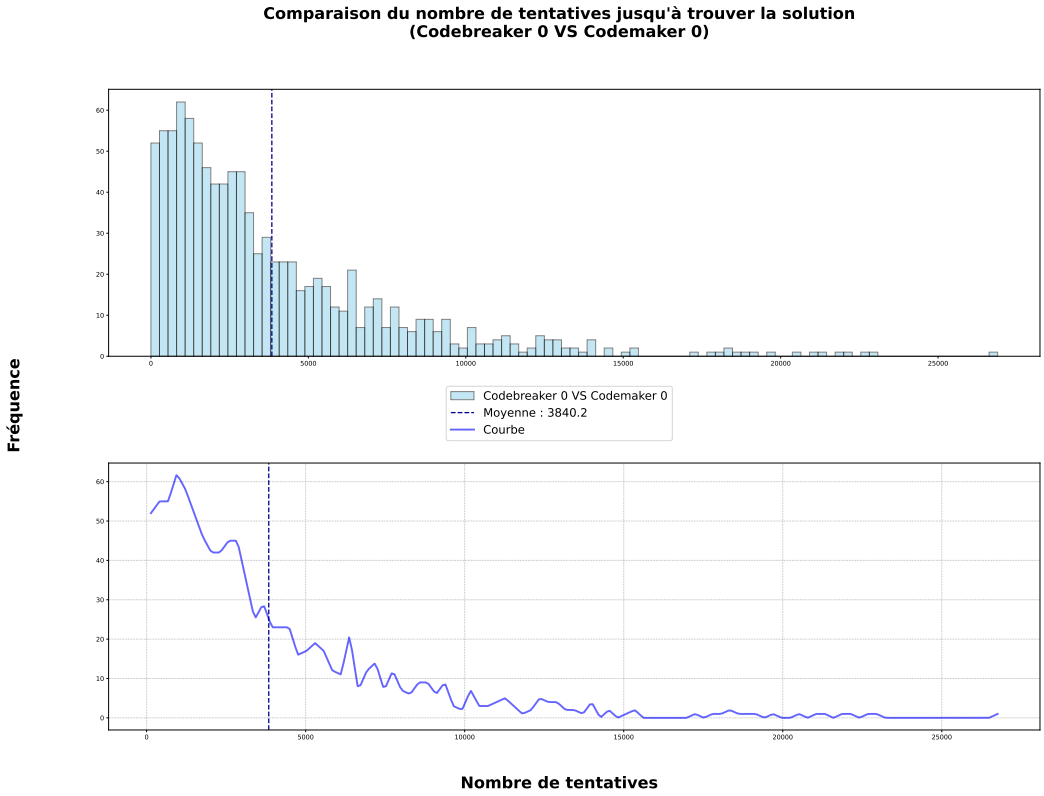



Figure 1 – Généré avec *histogram.py*

## 2.4 Question 4

Objectif

Ecrire une fonction `codebreaker(évaluation)` dans *codebreaker1.py*, capable de sélectionner aléatoirement l'une des combinaisons qui n'a pas encore été testées, cela afin d'améliorer l'efficacité de la recherche de la solution.

Explications du code

1. **Initialisation du *codebreaker* :**

   A chaque début de partie, le *codebreaker* génère et enregistre dans une variable globale, un ensemble qui contiendra les combinaisons déjà testées.
2. **Choix de la combinaison :**

On génère aléatoirement une combinaison jusqu'à en trouver une qui n'a pas encore été testées. Elle est ensuite ajoutée à l'ensemble des combinaisons déjà essayées pour ne pas la re-choisir lors des tentatives futures.

3. **Retour des résultats :**

   On renvoie la combinaison choisit.

EXPLICATIONS

**Essaies jusqu'à trouver la solution :**

Notons $X_k$ la variable aléatoire réelle telle que $X_k = 1$ si *codebreaker* a trouvé la bonne combinaison à la k-ième tentative, $X_k = 0$ sinon.

Notons $N$ la variable aléatoire réelle qui compte le nombre de tentatives faites pour trouver la bonne combinaison.

Soit $k \in \left\{1, \dots, COLORS^{LENGTH}\right\}$

$$\begin{aligned}\{N = k\} &= \underbrace{\{X_1 = 0\} \cap \cdots \cap \{X_{k-1} = 0\}}_{k-1 \text{ fois}} \cap \{X_k = 1\} \\ &= \bigcap_{i=1}^{k-1} \{X_i = 0\} \cap \{X_k = 1\}\end{aligned}$$

D'après la formule des probabilités composées :

$$\begin{aligned}\mathbb{P}(N = k) &= \mathbb{P}(X_1 = 0)\mathbb{P}_{\{X_1=0\}}(X_2 = 0) \times \cdots \times \mathbb{P}_{\{X_1=0\}\cap\cdots\cap\{X_{k-1}=0\}}(X_k = 1) \\ &= \prod_{i=0}^{k-2} \left(\frac{COLORS^{LENGTH} - (i+1)}{COLORS^{LENGTH} - i}\right) \frac{1}{COLORS^{LENGTH} - (k-1)} \\ &= \frac{COLORS^{LENGTH} - (k-1)}{COLORS^{LENGTH}} \frac{1}{COLORS^{LENGTH} - (k-1)} \quad \text{Produit télescopique} \\ &= \frac{1}{COLORS^{LENGTH}}\end{aligned}$$

$N$ suit une loi uniforme car :

- $N(\Omega) = \left\{1, \dots, COLORS^{LENGTH}\right\}$
- $\forall k \in \left\{1, \dots, COLORS^{LENGTH}\right\}, \mathbb{P}(N = k) = \frac{1}{COLORS^{LENGTH}}$

**Calcul de l'espérance :**

$$
\begin{aligned}
E(N) &= \sum_{k=1}^{COLORS^{LENGTH}} |k\mathbb{P}(N=k)| \\
&= \sum_{k=1}^{COLORS^{LENGTH}} \left[k \times \frac{1}{\text{COLORS}^{\text{LENGTH}}}\right] \\
&= \frac{1}{\text{COLORS}^{\text{LENGTH}}} \times \left[\text{COLORS}^{\text{LENGTH}} \times \frac{COLORS^{LENGTH}+1}{2}\right] \\
&= \frac{COLORS^{LENGTH}+1}{2} \\
&= \frac{8^4+1}{2} \\
&= 2048.5
\end{aligned}
$$

Cela signifie que pour un très grand nombre de partie jouées, on a en moyenne trouvé la solution au bout de 2049 tentatives.

Afin de vérifier notre approche théorique, nous avons simulé un grand nombre de parties (100 000), et enregistré le nombre d'essaies nécessaires au *codebreaker* pour trouver la solution.

## Tests & Analyses

En affichant les résultats d'une simulation de 1000 parties pour le *codebreaker 0* et le *codebreaker 1*, on peut comparer l'efficacité de ces deux stratégies.

Le graphique montre :

- Pour le *codebreaker 1*, le nombre de tentatives nécessaires ne dépasse pas $\simeq 4096$, ce qui est cohérent puisqu'on n'essaie que des combinaisons non déjà testées, et qu'il y a 4096 possibilités.
- Avec le *codebreaker 1*, $\forall k \in \mathbb{N}^*, \mathbb{P}(N=k) \simeq cste$
  Ce qui nous confirme que $N$ suit une loi uniforme.
- L'espérance observée, avoisinant 2056, reste proche de la valeur théorique 2048.5, ce qui valide notre modèle analytique.

  Cependant, ce modèle n'est toujours pas représentatif de la réalité puisqu'on n'a pas autant de tentatives. Dans la suite, nous verrons d'autres stratégies pour réduire cette espérance.
- L'espérance observée pour le *codebreaker 0* ($\simeq 4096.8$) est deux fois supérieur à celle pour le *codebreaker 1* ($\simeq 2056$), le *codebreaker 1* est donc plus efficace.

  C'est cohérent puisque cette fois-ci, chaque combinaison n'est testée qu'une seule fois au maximum.

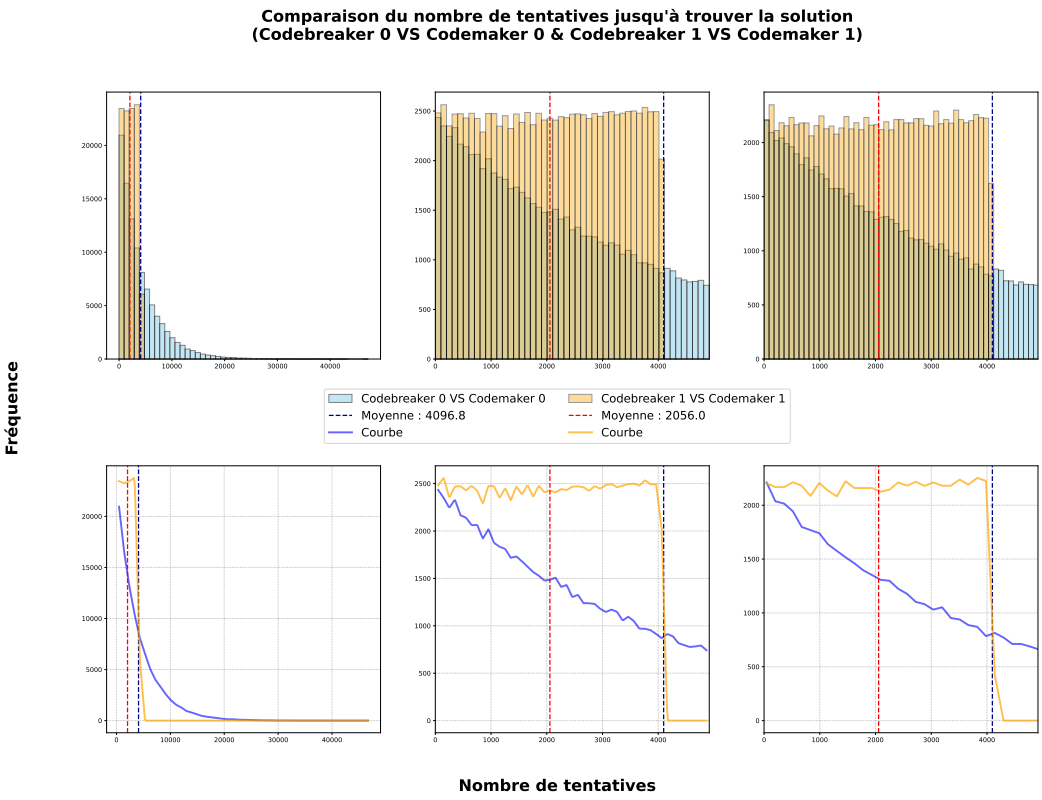



FIGURE 2 – Généré avec *histogram.py*

## 2.5 Question 5

OBJECTIF

Ecrire une fonction `donner_possibles(combinaison, évaluation)` dans *common.py*, qui détermine l'ensemble des combinaisons encore possibles après une tentative du *codebreaker*.

EXPLICATIONS DU CODE

1. **Initialisation des variables :**

   Au début de chaque partie, un ensemble vide est créé pour stocker les solutions possibles, ainsi qu'un autre ensemble contenant toutes les permutations envisageables.

2. **Première mise à jour des solutions possibles :**

   Lors de la première mise à jour, seules les permutations dont l'évaluation par rapport à la première combinaison testée, est identique à celle fournie en paramètre, sont conservées.

3. **Retour des résultats :**

   On renvoie l'ensemble des solutions possibles.

TESTS & ANALYSES

Nous avons fait quelques tests pour certaines valeurs et avec des *COLORS* et *LENGTH* réduits, les résultats étaient cohérents. La fonction est relativement simple, nous nous assurerons du bon fonctionnement de cette fonction lorsqu'on l'utilisera dans d'autres fonctions.

```python
# IL FAUT BIEN RECOMMENTER TOUTES LES LIGNES DE LA PARTIE TESTS
    UNE FOIS LES TESTS FINIS !!!

LENGTH = 3
COLORS = ["R", "G", "B"]

# On doit obtenir {'BRG', 'GBR', 'RGB'}
print(donner_possibles("RBG", (1, 2)))

# On doit obtenir {'RGG', 'BBG', 'RBR', 'RBB', 'GBG', 'RRG'}
print(donner_possibles("RBG", (2, 0)))

# On doit obtenir set()
print(donner_possibles("RBG", (2, 1)))
```

## 2.6 Question 6

Objectif

Ecrire une fonction `maj_possibles(solutions possibles, combinaison, évaluation)` dans *common.py*, qui met à jour l'ensemble des solutions encore possibles après qu'une nouvelle combinaison soit testée.

Explications du code

1. **Mises à jour des solutions possibles** :

   À chaque tentative, on garde uniquement les solutions possibles dont l'évaluation par rapport à la combinaison testée correspond à celle obtenue en comparant cette combinaison à la solution réelle.

Tests & Analyses

Nous avons fait quelques tests pour certaines valeurs données en paramètres, les résultats étaient cohérents. La fonction est relativement simple, nous nous assurerons du bon fonctionnement de cette fonction lorsqu'on l'utilisera dans d'autres fonctions.

```python
# IL FAUT BIEN RECOMMENTER TOUTES LES LIGNES DE LA PARTIE TESTS
    UNE FOIS LES TESTS FINIS !!!

LENGTH = 3
COLORS = ["R", "G", "B"]

solutions_possibles = get_permutations()
maj_possibles(solutions_possibles, "RBG", (1, 2))
print(solutions_possibles) # On doit obtenir {'BRG', 'GBR', 'RGB'}
```

## 2.7 Question 7

Objectif

Ecrire une fonction `codebreaker(évaluation)` dans *codebreaker2.py*, capable de tester une combinaison parmi les solutions possibles.

Explications du code

1. **Initialisation du *codebreaker* :**

   Au début de chaque partie, le *codebreaker* crée et stocke dans une variable globale l'ensemble des solutions possibles, ainsi qu'une variable pour la dernière combinaison testée.

2. **Choix de la combinaison :**

   - Si c'est la première tentative du *codebreaker*, il choisit aléatoirement une combinaison.
   - Sinon, il met à jour l'ensemble des solutions possibles en tenant compte de l'évaluation de la tentative précédente. Puis il sélectionne aléatoirement une combinaison parmi les solutions encore possibles.

3. **Retour des résultats :**

   On renvoie la combinaison choisit.

Tests & Analyses

En affichant les résultats d'une simulation de 1000 parties pour le *codebreaker 1* face au *codemaker 1* et le *codebreaker 2* face au *codemaker 1*, on peut comparer l'efficacité de ces deux stratégies.

Le graphique montre :

- Pour le *codebreaker 2*, le nombre de tentatives requises est très bas, avoisinant $\simeq 10$, tandis que pour le *codebreaker 1*, il peut parfois atteindre jusqu'à $\simeq 4100$.
- L'espérance observée pour le *codebreaker 2*, avoisinant 5.5, est nettement inférieure à celle du *codebreaker 1*, avoisinant 2071.3. La version 2 est donc bien plus efficace.

  Cette fois, le modèle est plus représentatif de la réalité, puisque généralement, on dispose d'une dizaine de tentatives.

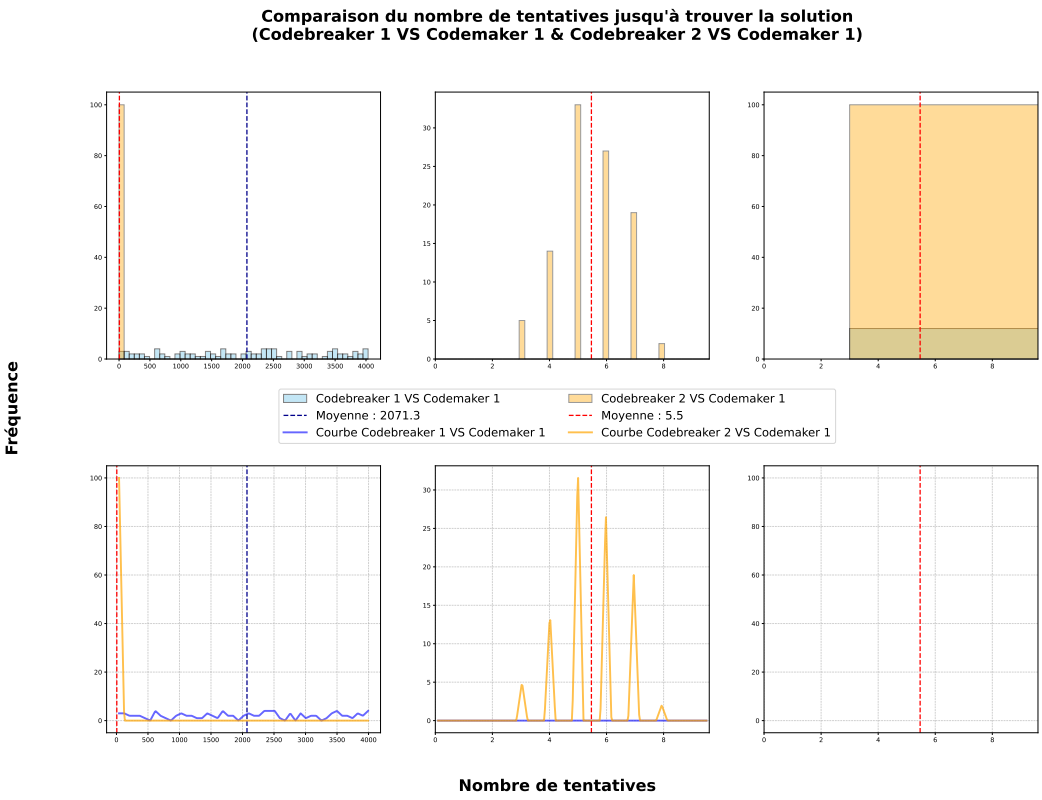



FIGURE 3 – Généré avec *histogram.py*

## 2.8 Question 8

OBJECTIF

Ecrire une fonction `codemaker(combinaison)` dans *codemaker2.py*, capable de changer de solution pendant le tour, tout en respectant les évaluations précédentes. Le but est de rendre la partie aussi longue que possible.

EXPLICATIONS DU CODE

1. **Initialisation du *codemaker* :**

   Au début de chaque partie, le *codemaker* génère aléatoirement la combinaison secrète et l'enregistre dans une variable globale. On défini également l'ensemble des solutions possibles.

2. **Principe de mémoization :**

   Afin de réduire la complexité de notre programme, nous avons développé un système de mémoization pour récupérer l'évaluation d'une combinaison par rapport à une autre.

   - Si l'évaluation a déjà été calculée, on renvoie la valeur enregistrée.
   - Sinon, on calcul, enregistre, puis renvoie cette évaluation.

3. **Evaluation :**

   Le but est de changer la solution à deviner en choisissant celle qui maximise le nombre de solutions possibles lorsqu'une combinaison est testée.

   - Si le nombre de solutions possibles est trop élevé ($> 1\,000\,000$), les parcourir

intégralement pour sélectionner la plus stratégique serait trop coûteux en termes de complexité. C'est pourquoi on ne change pas de solution, on met seulement à jour l'ensemble des solutions possibles.

- Dans le cas contraire, on examine chaque solution possible et on détermine combien resteraient valides après un changement de solution. Finalement, on sélectionne celle qui permet d'augmenter au maximum le nombre de solutions possibles.

On peut alors mettre à jour l'ensemble des solutions possibles après avoir joué une combinaison et avoir éventuellement modifié la solution.

4. **Retour des résultats** :

   On renvoie l'évaluation de la combinaison testée par rapport à la solution éventuellement modifiée.

Tests & Analyses

En affichant les résultats d'une simulation de 100 parties pour le *codebreaker 2* face au *codemaker 2* et le *codebreaker 2* face au *codemaker 1*, on peut comparer l'efficacité de ces deux stratégies.

Le graphique montre :

- Le nombre de tentatives requises pour trouver la solution est assez bas, que ce soit pour le *codemaker 2* ou pour le *codemaker 1*.
- L'espérance observée pour le *codemaker 2*, avoisinant 6.7, est légèrement supérieure à celle du *codemaker 1*, avoisinant 5.5. La version 2 est donc légèrement plus efficace.

  Ceci est cohérent puisque le *codemaker 2* modifie la solution en cours de jeu pour rallonger la partie.

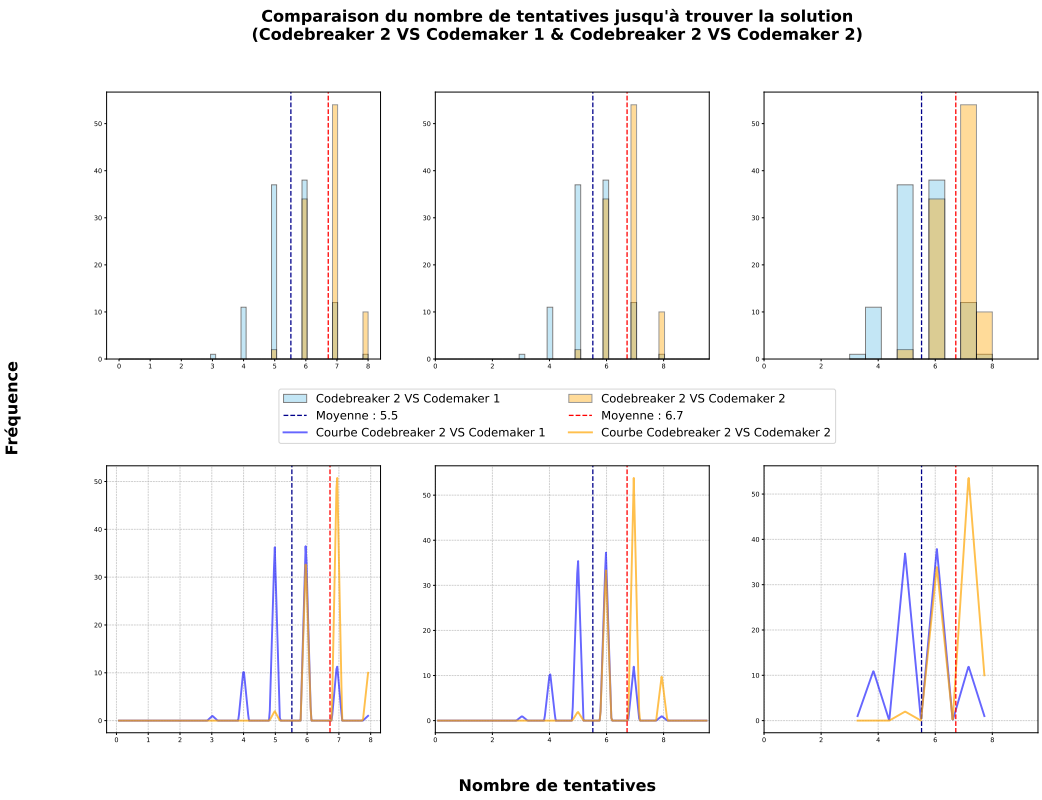



FIGURE 4 – Généré avec *histogram.py*

## 2.9 Question 9

OBJECTIF

Ecrire une fonction `play_log(codemaker, codebreaker, nom fichier)` dans *play.py*, qui enregistre le déroulement (combinaisons testées et leur évaluation associée) d'une partie dans un fichier texte.

EXPLICATIONS DU CODE

1. **Fonctionnement d'une partie** :

   Les fonctionnement d'une partie est identique à celui de la fonction `play(...args)` donnée.
2. **Enregistrement de la partie** :

   On ouvre un fichier de logs, dans lequel on écrit la combinaison testée, suivi d'un retour à la ligne, puis de l'évaluation associée.
3. **Retour des résultats** :

   On renvoie le nombre de tentatives nécessaires.

TESTS & ANALYSES

Pour tester cette fonction, nous avons simulé une partie en affichant progressivement les tentatives effectuées ainsi que leur évaluation correspondante. Ensuite, nous avons vérifié la cohérence de ces résultats et leur mise en page dans le fichier de logs.

## 2.10 Question 10

Objectif

Ecrire une fonction `check_codemaker(nom fichier)` dans *check_codemaker.py*, qui à partir des logs, vérifie que le codemaker n'a pas triché.

Explications du code

1. **Initialisation des variables :**

   On ouvre le fichier de logs, et on récupère dans une liste les combinaisons testées et leur évaluation associée.
   Si la partie n'est pas terminée, on n'a pas accès la solution finale, on ne peut donc pas vérifier si le codemaker a triché ou non
2. **Vérifications :**

   Pour chacune des tentatives, on regarde si l'évaluation de la combinaison testée provenant des logs, est identique à l'évaluation de cette combinaison par rapport à la solution. Si ce n'est pas le cas, le codemaker a triché.
3. **Retour des résultats :**

   On renvoie le nombre de tentatives nécessaires.

Tests & Analyses

```
1  import play
2  import codemaker2, codebreaker2
3
4  # Simuler une partie :
5  play.play_log(codemaker2, codebreaker2, "log0.txt")
6
7  # Verifier que le codemaker n'a pas triche de maniere visible :
8  print(check_codemaker("log0.txt"))
9
10 # Re-commenter les deux lignes de code precedentes.
11 # Modifier l'une des evaluations dans le fichier "log0.txt", puis
       verifier que le codemaker a triche de maniere visible :
12 print(check_codemaker("log0.txt")) # A DECOMMENTER
```

## 2.11 Question 11

Objectif

Vérifier s'il n'est pas plus avantageux, dans certaines situations, de tester une combinaison que l'on sait déjà impossible afin d'obtenir d'avantage d'informations.

Explications

Intuitivement, on pourrait penser que tester une combinaison que l'on sait déjà faux, ne sert à rien, puisqu'elle sera faux dans tous les cas. Par exemple si l'on teste *RRRR* et que l'évaluation est *(0,0)*, alors quel est l'intérêt d'essayer une combinaison contenant du rouge ?

Prenons un exemple :

On considère qu'on a :

- `LENGTH=3`

- `COLORS=['R', 'V', 'B', 'J', 'N']`

En simulant une partie où le *codebreaker 2* joue contre le *codemaker 1*, on obtient les logs suivants :

```
import codebreaker2, codemaker1

play_log(codemaker1, codebreaker2, "log0.txt")
```

```
Essai 1 : VVV (0,0)
Solutions possibles :
{'RRJ', 'BNR', 'NJR', 'NJB', 'BRR', 'NBR', 'NNN', ..., 'NBJ'}

Essai 2 : JNJ (2,0)
Solutions possibles :
{'NNJ', 'JNN', 'JJJ', 'JNR', 'JBJ', 'JRJ', 'BNJ', 'RNJ', 'JNB'}

Essai 3 : JNB (2,0)
Solutions possibles :
{'JNN', 'JNR'}

Essai 4 : JNR (2,0)
Solutions possibles :
{'JNN'}

Essai 5 : JNN (3,0)
Bravo ! Trouve JNN en 5 essais
```

Cependant, si, à l'essaie 3, on avait testée $NNV$ que l'on sait impossible, on aurait eu le déroulement suivant :

```
Essai 1 : VVV (0,0)
Solutions possibles :
{'RRJ', 'BNR', 'NJR', 'NJB', 'BRR', 'NBR', 'NNN', ..., 'NBJ'}

Essai 2 : JNJ (2,0)
Solutions possibles :
{'NNJ', 'JNN', 'JJJ', 'JNR', 'JBJ', 'JRJ', 'BNJ', 'RNJ', 'JNB'}

Essai 3 : NNV (1,1)
Solutions possibles :
{'JNN'}

Essai 4 : JNN (3,0)
Bravo ! Trouve JNN en 4 essais
```

L'essai 1 nous dit que la lettre $V$ n'est pas présente dans la solution. Donc, l'évaluation de l'essai 3 nous indique que les deux $N$ doivent nécessairement être placés à la fin de la combinaison. De plus, grâce à l'essai 2, nous savons que la lettre $J$ doit être en première position.

En utilisant une combinaison que nous savions impossible, nous avons donc pu réduire le nombre total d'essais nécessaires pour trouver la solution finale. Cette stratégie est donc intéressante.

## 2.12 Question 12

Objectif

> Ecrire une fonction `codebreaker(evaluation)` dans *codebreaker3.py* qui teste une combinaison de manière à minimiser, dans le pire des cas, l'ensemble des solutions possibles si le *codemaker* décide de changer de solution.

Explications du code

> 1. **Initialisation du *codebreaker* :**
>
>    Au début de chaque partie, le *codebreaker* crée et stocke dans une variable globale l'ensemble des solutions possibles, ainsi qu'une variable pour la dernière combinaison testée.
> 2. **Choix de la combinaison :**
>
>    - Si c'est la première tentative du *codebreaker*, il choisit aléatoirement une combinaison.
>    - Sinon, il met à jour l'ensemble des solutions possibles en tenant compte de l'évaluation de la tentative précédente. Puis pour chaque permutation qu'on peut tester, on récupère le nombre de solutions possibles dans le pire des cas si le *codemaker* change de solution. Puis on sélectionne la permutation qui minimise ce pire des cas.
> 3. **Retour des résultats :**
>
>    On renvoie la combinaison choisit.

Tests & Analyses

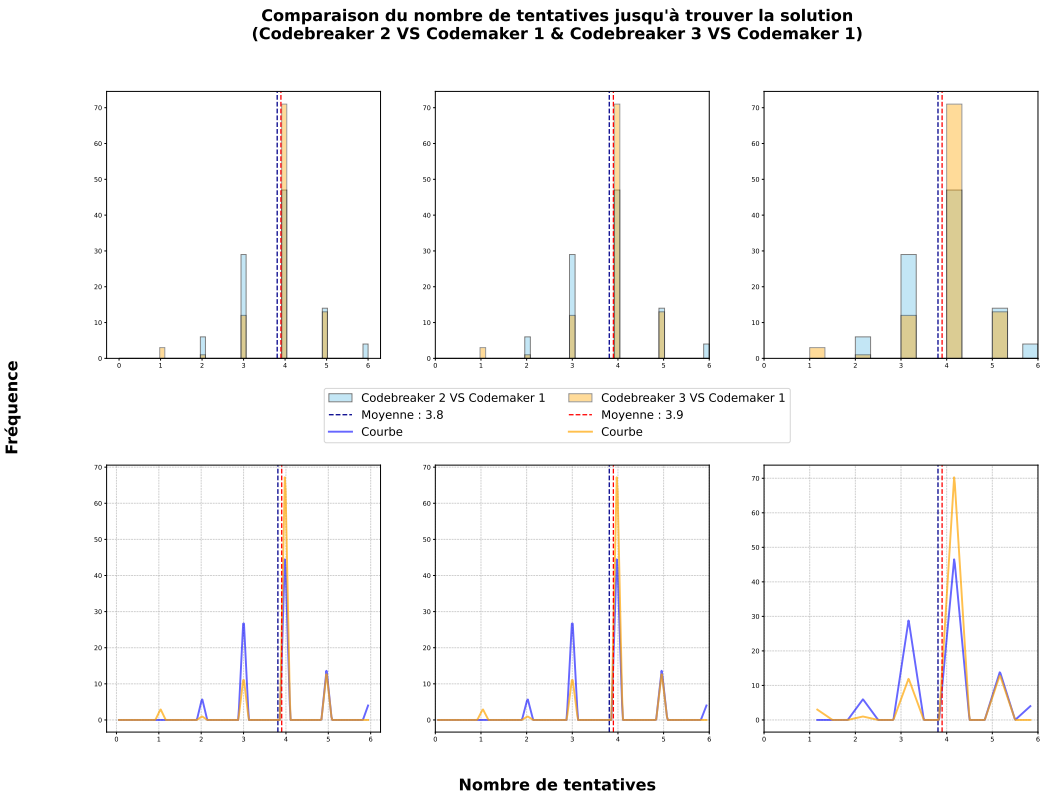



FIGURE 5 – Généré avec *histogram.py*

En affichant les résultats d'une simulation de 200 parties pour le *codebreaker 2* face au *codemaker 1* et le *codebreaker 3* face au *codemaker 1*, on peut comparer l'efficacité de ces deux stratégies.

Le graphique montre :

- Le nombre de tentatives requises pour trouver la solution est assez bas, que ce soit pour le *codebreaker 2* ou pour le *codebreaker 3*.
- L'espérance observée pour le *codebreaker 2*, avoisinant 3.8, est très proche de celle du *codebreaker 3*, avoisinant 3.9.

  Ceci est cohérent puisque le *codemaker 1* ne change pas de solution pour rallonger la partie, donc le codebreaker 3 est inefficace puisqu'il anticipe inutilement, le pire rallongement de partie possible. Au contraire, le *codebreaker 3* est légèrement moins efficace que le *codebreaker 2* dans ce cas là, puisqu'il test la solution qui minimise ce pire des cas : il peut alors tester des combinaisons qui apportent moins d'information pour rien (ce qui explique l'espérance légèrement plus élevée).

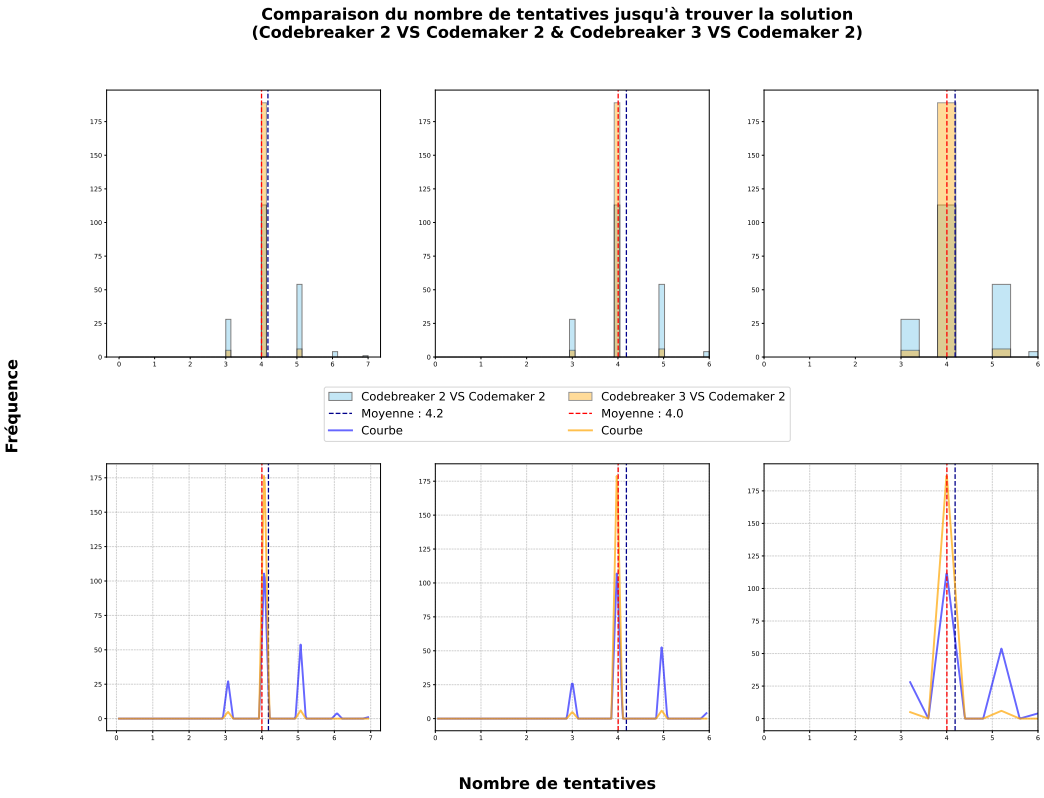



FIGURE 6 – Généré avec *histogram.py*

En affichant les résultats d'une simulation de 200 parties pour le *codebreaker 2* face au *codemaker 2* et le *codebreaker 3* face au *codemaker 2*, on peut comparer l'efficacité de ces deux stratégies.

Le graphique montre :

- Le nombre de tentatives requises pour trouver la solution est assez bas, que ce soit pour le *codebreaker 2* ou pour le *codebreaker 3*.
- L'espérance observée pour le *codebreaker 2*, avoisinant 4.2, est légèrement supérieure à celle du *codebreaker 3*, avoisinant 4.0. La version 3 est donc légèrement plus efficace face au *codemaker 1*.

  Ceci est cohérent puisque le *codebreaker 3* anticipe le changement de solution que pourra faire *codemaker 2*, alors que le *codebreaker 2* se contente seulement de tester une combinaison non déjà testée.

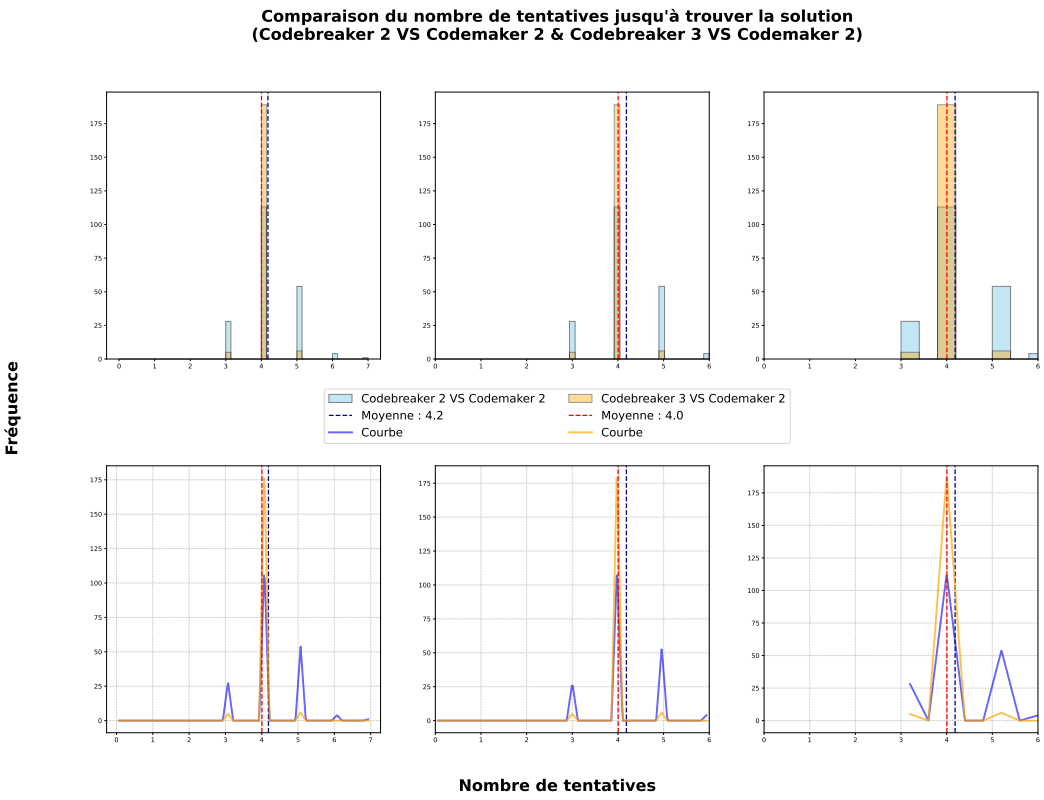



FIGURE 7 – Généré avec *histogram.py*

## 2.13 Question 13

EXPLICATIONS DU CODE

Nous ne détaillerons pas le fonctionnement de l'interface graphique, car des explications détaillées sont disponibles dans les fichiers du code. En revanche, nous allons décrire la structure de notre programme.

1. **Initialisation de l'application** :

   Lors de l'execution du fichier */gui/main.py*, une fenêtre est créée ainsi qu'une instance de la classe `Mastermind` , qui dirige l'utilisateur vers la page de configuration de la partie.

2. **Configuration de la partie** :

   La classe `Mastermind` génère l'interface graphique permettant de définir les paramètres du jeu. Il est possible de :

   - Sélectionner le *codemaker* contre lequel jouer.
   - Choisir le nombre de couleurs disponibles.
   - Définir la taille de la solution.
   - Définir le nombre de tentatives possibles.

   Enfin, un bouton permet de créer une instance de la classe `Game` pour lancer le jeu.

3. **Lancement de la partie** :

(a) **Initialisation du *codemaker* :**

Au début de la partie, le *codemaker* est initialisé pour générer la combinaison secrète à deviner.

(b) **Grille de combinaisons :**

Pour $n$ tentatives, on génère $n$ instances de la classe `Attempt` .
Chaque instance possède plusieurs propriétés, notamment :

- `activated` : représente le plot actuellement sélectionné (par défaut, le premier).
- `currentAttempt` : vaut *True* si la tentative en cours est active, sinon *False*.

(c) **Barre de couleurs :**

La barre de $k$ couleurs contient les $k$ couleurs disponibles pour composer une combinaison. Lorsqu'on clique sur une couleur ou qu'on appuie sur un chiffre appartenant à $[1, k]$, la couleur correspondante est attribuée au plot sélectionné.

4. **Evaluation d'une combinaison :**

Si la combinaison n'est pas complète, on renvoie une erreur.
Sinon le *codemaker* évalue la combinaison testée, et affiche le nombre de plots bien placés ainsi que ceux présents, mais mal placés.
A côté de chaque combinaison testée, on affiche un carré avec vert donnant le nombre de plots bien placés, ainsi qu'un rouge donnant le nombre de plots présents, mais mal placés.

5. **Fin de partie :**

A la fin de la partie, la fenêtre se ferme et une nouvelle s'ouvre. Le joueur peut alors rejouer.

# 3 Conclusion

Ce projet nous a permis d'étudier l'importance des différentes stratégiques à travers les différentes versions du *codemaker* et du *codebreaker*.

De manière générale, plus un programme cherche à optimiser la résolution du problème (*codebreaker*) ou l'évaluation des combinaisons (*codemaker*), plus son temps d'exécution augmente. Il est donc essentiel de trouver un équilibre entre la réduction du nombre de tentatives et la rapidité avec laquelle la solution est obtenue.

Par ailleurs, une approche alternative aurait été de pré-générer un fichier contenant toutes les évaluations possibles pour chaque combinaison, évitant ainsi de les recalculer à chaque exécution. Mais ça aurait été beaucoup trop long... C'est pourquoi nous avons choisi de limiter l'optimisation à une mémoization sur une partie jouée, afin de gagner en rapidité.

On a pris beaucoup de plaisir sur ce projet, puisqu'il est relativement concret avec des analyses à chaque question. De plus, on est plutôt content de notre interface graphique sachant qu'on en avait jamais fait auparavant.