Informatique (S4) - Projet Mastermind

Équipe pédagogique Info CPP S4

2026

Ce projet est à réaliser en équipes de 2 étudiants formées au sein d'un même groupe de TD [1]. Les questions sont à traiter en parallèle de l'implémentation. Le rendu final consiste en une archive `Nom1_Nom2_lettregroupeTP.zip` au format zip contenant vos programmes en langage Python et votre compte-rendu au format pdf. Le non-respect du format de l'archive sera sanctionné. Cette archive est à rendre au plus tard pour le dimanche 24 avril 2026 - 23h59 sur Moodle. Ce projet compte pour 1/4 de la note en informatique du semestre.

Prêtez une attention particulière à la rédaction du compte-rendu et à la mise en forme des graphiques. On attend une analyse critique (et non une simple description) des résultats de vos programmes. Tout le code écrit pour répondre aux questions doit être rendu et documenté. Vos fonctions et programmes doivent être testés systématiquement ; ceci sera pris en compte dans l'évaluation. **Vos tests devront être réalisés dans un ou plusieurs fichiers dédiés et ne devront pas faire partie des fichiers contenant le code demandé.**

La copie ou le partage de programmes ou de réponses avec un autre binôme ainsi que l'utilisation de matériel trouvé sur internet sont strictement interdits et seront sévèrement sanctionnés.

Deux séances de suivi de ce projet sont prévues. Elles nous permettront d'évaluer votre avancement et de répondre à vos questions éventuelles. Elles se dérouleront :

— Séance 1 : le mardi 10 février,
— Séance 2 : le mardi 17 mars.

Pour que la 1re séance de suivi soit productive, il faut venir en ayant commencé à travailler sur le projet afin de voir les points qui vous bloquent. La 2e séance servira à faire un bilan pour organiser la préparation du rendu.

Introduction

Vous connaissez probablement le jeu du mastermind. Si ce n'est pas le cas, familiarisez-vous avec ce jeu en lisant ce tutoriel et en jouant contre un ordinateur (par exemple ici).

Le but de ce projet est d'implémenter en Python des programmes qui jouent le *codebreaker* et le *codemaker*, avec une stratégie plus ou moins intelligente pour le premier et plus ou moins honnête pour le second.

Détails techniques, structure des programmes, exemple minimal

Une combinaison est une chaine de `LENGTH` caractères, les caractères autorisés étant listés dans `COLORS`. Une même couleur peut être présente plusieurs fois.

Le *codemaker* choisit la combinaison à faire deviner. Pour chaque combinaison proposée par le *codebreaker*, il donne l'évaluation de cette combinaison. L'évaluation est un couple d'entiers, indiquant respectivement le nombre de plots de la combinaison proposée présents dans la solution à la bonne position, et le nombre de plots présents mais à une mauvaise position.

Le *codebreaker* propose des combinaisons, choisies en fonction des évaluations précédemment

1. Éventuellement quelques équipes de 3 **après validation par l'enseignant** en cas de groupes de TD impairs.

données par le *codemaker*.

On fournit l'architecture globale du jeu, avec un embryon de codebreaker et de codemaker. Voici le contenu des principaux fichiers fournis :

- `common.py` contient la définition des variables `LENGTH` et `COLORS`. On y ajoutera (en suivant rigoureusement l'énoncé) des fonctions qui peuvent être utilisées à la fois par le codemaker et le codebreaker.
- `codemaker0.py` est une implémentation *partielle* du codemaker, qui ne compte que les plots présents à la bonne position.
- `codebreaker0.py` est une implémentation rudimentaire du codebreaker qui essaie des combinaisons au hasard jusqu'à trouver la bonne.
- `play.py` lance ces versions 0 du codemaker et du codebreaker et les fait jouer ensemble, en utilisant une fonction `play` qui peut faire jouer n'importe quelle version du codemaker contre n'importe quelle version du codebreaker. Ce programme peut facilement être modifié pour faire jouer ensemble d'autres versions du codebreaker et du codemaker. Pour vous aider à tester vos programmes, il contient aussi du code pour faire jouer un humain (dont les propositions / corrections sont lues au clavier) dans le role du codebreaker ou du codemaker.

Il est nécessaire de bien comprendre ce fonctionnement avant d'attaquer la suite du projet. Si ce n'est pas le cas, demandez de l'aide à vos enseignants.

L'essentiel du projet consistera en l'implémentation de versions plus avancées du codemaker et codebreaker. **Il est essentiel de respecter parfaitement les spécifications** (noms des programmes, noms des fonctions et leurs arguments, etc) imposées par l'énoncé. Vos nouvelles versions pourront ainsi s'interfacer avec les versions 0 données du codemaker et codebreaker (vous pouvez utiliser et adapter le code proposé dans ces versions 0). Notamment, vous ne pouvez pas changer les arguments des fonctions `codebreaker / codemaker` et `init`, qui sont les seules fonctions à travers lesquelles la boucle principale de jeu dans `play.py` communique avec le codemaker et le codebreaker. La fonction `play` ne doit donc pas être modifiée pour faire fonctionner les codebreaker et codemaker que vous écrirez dans ce projet.

Auto-tests

Pour vous aider à respecter les spécifications et vous éviter des erreurs importantes dès les premières questions, on fournit un script d'auto-test `selftest.py`, à placer dans le répertoire contenant vos fichiers python et à lancer sans arguments. Ce script vérifie que certaines fonctions clés (questions 1 à 5) sont présentes dans le bon fichier et renvoient le résultat attendu sur quelques exemples simples.

Le succès de ces autotests ne garantit en aucun cas que votre travail est correct (et mettre en place des tests plus avancés fait partie des réflexes d'un bon développeur). En revanche, un échec aux auto-tests indique un problème majeur dans votre code. **L'échec des autotests sera sévèrement sanctionné, donc assurez-vous que les autotests fonctionnent**.

Première version fonctionnelle

La version 0 du codemaker n'est pas entièrement fonctionnelle puisqu'elle ne compte pas les plots existants mais mal placés.

Question 1 — Écrivez dans `common.py` une fonction `evaluation` qui évalue une combinaison donnée en premier argument en la comparant à la combinaison de référence donnée en deuxième

argument [2]. L'évaluation est renvoyée comme un couple de deux entiers (nombre de plots présents bien et mal placés, dans cet ordre).

Question 2 — Implémentez (dans `codemaker1.py`) une version 1 du codemaker qui évalue correctement les combinaisons proposées en s'appuyant sur la fonction `evaluation` que vous avez écrite dans le fichier `common.py`.

Un peu de mémoire

La version 0 du codebreaker n'est pas très efficace puisqu'elle essaie des combinaisons aléatoires. Avant de commencer à l'améliorer, commençons par quantifier son efficacité.

Question 3 — Quelle est l'espérance du nombre d'essais faits par cette version avant de trouver la bonne solution (en fonction de `LENGTH` et de la longueur de `COLORS`) ? Lancez un grand nombre de parties et produisez un histogramme du nombre d'essais nécessaires [3] (pour les valeurs de `LENGTH` et `COLORS` définies dans `common.py`), comparez avec votre résultat analytique.

En tirant des combinaisons aléatoires, il est probable que le codebreaker essaie plusieurs fois la même combinaison.

Question 4 — Quel gain attendez-vous (en terme d'espérance du nombre d'essais faits pour trouver la bonne solution) si le codebreaker choisit aléatoirement une combinaison *pas encore essayée* ? Implémentez (dans `codebreaker1.py`) cette version 1 du codebreaker et testez votre prédiction : produisez un graphique de votre choix permettant de visualiser le gain en nombre d'essais [4].

Pour avoir une mémoire entre différents appels à la fonction `codebreaker`, vous pouvez utiliser des variables globales, de la même façon que le codemaker utilise la variable globale `solution` pour se souvenir de la solution choisie. Il est important de bien comprendre le fonctionnement des variables globales (et ce qui se passerait si ces variables n'étaient pas déclarées globales). Si nécessaire, on propose des exemples simples dans `aide_variables_globales.py`.

Un embryon de logique

Même en gardant en mémoire les combinaisons déjà essayées, l'essai par force brute [5] est très inefficace. Avec 4 plots et 8 couleurs, un joueur humain non expert est généralement capable de trouver la combinaison en une dizaine d'essais. Pour cela une stratégie simple mais efficace consiste à n'essayer que des combinaisons possibles au vu des évaluations des combinaisons précédemment proposées.

Question 5 — Écrivez dans `common.py` une fonction `donner_possibles` qui détermine l'ensemble des combinaisons possibles (renvoyé comme une variable de type `set`) après un unique essai (les arguments sont la combinaison testée et l'évaluation associée, dans cet ordre).

Chaque étape (proposition d'une solution et obtention d'une évaluation) supprime un certain nombre de combinaisons précédemment possibles mais n'ouvre jamais de nouvelles possibilités. Le

2. N'oubliez pas de tester en profondeur cette fonction. Une erreur dans cette fonction de base sera probablement la cause de programmes faux pour tout le reste du sujet.
3. Le code utilisé pour réaliser ce grand nombre de parties et produire cet histogramme doit bien sûr être documenté et être inclus dans le rendu.
4. Choisissez le nombre de parties à lancer selon la rapidité de votre programme.
5. C'est comme ça qu'on appelle l'essai exhaustif de toutes les solutions.

codebreaker devra donc maintenir une variable de type `set` indiquant l'ensemble des combinaisons encore possibles.

Question 6 — Écrivez dans `common.py` une fonction `maj_possibles` qui met à jour cette variable, prenant en arguments, dans cette ordre, l'ensemble des combinaisons encore possibles avant ce nouvel essai, la combinaison testée, et l'évaluation associée. Plutôt que de renvoyer une nouvelle variable de type `set`, la variable donnée en premier argument sera directement modifiée [6].

Question 7 — Implémentez (dans `codebreaker2.py`) une version 2 du codebreaker qui n'essaie que des combinaisons possibles au vu des évaluations précédentes, en choisissant aléatoirement parmi toutes les possibilités. Cette version est-elle plus efficace (comparée à la version 1) ? Justifiez votre réponse avec un graphique.

On a maintenant une stratégie qui fonctionne relativement bien, similaire à la stratégie intuitivement adoptée par beaucoup de joueurs humains.

Un soupçon de rouerie

Dans la version analogique du jeu, le codemaker réalise physiquement sa combinaison sur le plateau de jeu [7]. Cela permet d'éviter que le codemaker ne change de combinaison en cours de jeu pour rallonger la partie.

Autorisons le codemaker à changer sa combinaison cible quand il le souhaite, à condition de respecter les évaluations déjà données [8]. L'objectif du codemaker est de faire durer la partie le plus longtemps possible.

Question 8 — Réalisez (dans `codemaker2.py`) une version 2 du codemaker aussi performante que possible, en laissant libre cours à votre imagination. Décrivez votre raisonnement et votre démarche [9]. Mesurez sa performance en la faisant jouer contre la version 2 du codebreaker, et produisez un graphique illustrant les performances de ce nouveau codemaker comparé à la version 1 (qui ne changeait pas de combinaison en cours de partie).

Analyse de parties

Au lieu (ou en plus) de l'affichage à l'écran, on aimerait pouvoir obtenir un *log* d'une partie lancée par `play.py` dans un fichier texte. Ce log sera une succession de lignes qui sont soit une proposition de combinaison soit une évaluation, en suivant strictement la syntaxe donnée dans l'exemple `log0.txt`.

Question 9 — Écrivez dans `play.py` une nouvelle version de la fonction `play` (que vous nommerez `play_log`) qui crée ce log dans un fichier texte dont le nom est fourni dans un troisième argument. Au contraire de `play`, votre fonction ne devra pas faire d'affichage à chaque étape.

Question 10 — Écrivez un programme `check_codemaker` qui à partir du log d'une partie terminée [10] (donné en argument) vérifie que le codemaker n'a pas triché de façon visible [11]. Votre

6. Si vous vous demandez comment une fonction peut modifier une variable donnée en argument, rappelez-vous que la situation est différente pour un entier ou pour des objets plus complexes comme `list` ou ici `set`.
7. En la gardant cachée pour le codebreaker jusqu'à la fin de la partie.
8. Autrement dit il peut tricher tant que le codebreaker ne peut pas s'en rendre compte.
9. N'hésitez pas à donner des exemples de parties pour illustrer votre propos.
10. Qui finit donc par l'évaluation `LENGTH,0`.
11. C'est à dire n'a pas changé de combinaison d'une façon qui invaliderait des évaluations précédemment données.

programme sera écrit dans un fichier `check_codemaker.py` qui prendra en argument un nom de ficher contenant un log de partie et affichera un message indiquant à quelle étape le codemaker a triché. Ainsi, si `log.txt` est un log de partie correcte, `check_codemaker.py log.txt` n'affichera rien.

Une dose d'optimal

Pour l'instant, notre codebreaker choisit une combinaison à tester aléatoirement parmi toutes les combinaisons possibles (au vu des évaluations précédentes).

Question 11 — Pensez-vous qu'il est toujours préférable pour le codebreaker de tester seulement des combinaisons encore possibles au vu des évaluations précédentes ? Autrement dit, pouvez-vous trouver une situation où essayer une combinaison qu'on sait déjà impossible donne plus d'information qu'essayer une des combinaisons encore possibles ? Le cas échéant, **donnez un exemple précis.**

L'idée d'un codebreaker optimal serait d'essayer la combinaison dont l'évaluation lui rapportera le plus d'informations *dans le pire des cas*. Ce pire des cas s'obtient notamment en considérant la rouerie maximale du codemaker qui de son coté cherche à donner le moins d'information possible.

Question 12 — Implémentez (dans `codebreaker3.py`) un tel codebreaker optimal. Faites le jouer contre les versions 1 et 2 du codemaker, commentez vos résultats (sans oublier de vous appuyer sur des graphiques pertinents) [12].

Une jolie interface

Pour utiliser votre programme, les utilisateurs s'attendent à avoir une interface intuitive, fonctionnelle, et si possible jolie (mais c'est un critère plutôt subjectif...). Afin de rendre plus attractive votre version du mastermind, vous allez implémenter une interface utilisateur graphique (GUI en anglais).

Pour cela, nous vous proposons d'utiliser la bibliothèque Tkinter. Ce n'est pas une obligation : si vous êtes plus à l'aise avec une autre bibliothèque d'interface graphique, vous pouvez l'utilisez à la place. Nous avons choisi Tkinter car elle nous semble être la plus simple.

Cette dernière partie du projet est intentionnellement très peu guidée. Quelques conseils néanmoins :

- N'essayez pas de comprendre à fond comment fonctionne Tkinter, certains concepts sont hors-programme (notamment les objets). Contentez-vous d'une compréhension pratique de cette bibliothèque, en allant voir des documentations et tutoriels. En revanche, vous devez comprendre (donc être capable d'expliquer) l'effet du code que vous utilisez.
- Écrivez votre interface de façon incrémentale, en testant l'interface après chaque modification de comportement.
- Limitez-vous au cas où le joueur prend le rôle du codebreaker. Si vous avez le temps et l'envie, vous pourrez ajouter d'autres possibilités (rôle du codemaker, deux joueurs) par la suite.
- Avant de commencer à programmer, décidez quel sont les éléments qui doivent apparaître sur l'interface et où ils doivent apparaître. Faire un schéma et un cahier des charges clair vous économisera beaucoup de temps par la suite. C'est d'autant plus vrai pour les modifications interactives de votre interface : écrivez clairement ce qui doit changer, quand

12. Si votre *codebreaker* est trop lent, vous pouvez diminuer dans `common.py` le nombre de couleurs possibles ou la longueur de la combinaison.

et comment.
— Vu la liberté qui vous est accordée dans cette partie, vous devez commenter encore davantage vos choix et votre programme.

Question 13 — Écrivez une interface graphique pour votre jeu dans le fichier `gui.py`.