Devoir noté – Mastermind Solver

Nicolas Voirol Jean-Cédric Chappelier

du 26 mars 2018, 12h00, au 11 avril 2018, 23h59.

I. Introduction et instructions générales

Ce devoir noté consiste à écrire une IA qui joue au jeu Mastermind. L'IA devra successivement proposer des combinaisons à l'utilisateur et se baser sur ses réponses pour déterminer la combinaison à laquelle le joueur pense. Dans ce devoir, nous vous demandons d'implémenter trois IA différentes, de plus en plus performantes :

1. une approche exhaustive (« *brute-force* ») qui teste chacune des combinaisons dans un ordre déterminé, jusqu'à trouver la bonne ;
2. une approche par élimination qui propose uniquement des combinaisons qui sont consistantes avec les réponses reçues pour les propositions précédentes ;
3. l'algorithme proposé par Donald Knuth lequel est garanti de trouver la solution en 5 tentatives ou moins (pour un Mastermind de taille 4).

Nous vous demandons d'écrire tout votre code dans *un seul* fichier nommé `mastermind.c` sur la base d'un fichier fourni, à compléter.

Indications : Si un comportement ou une situation donnée n'est pas définie dans la consigne ci-dessous, vous êtes libres de définir le comportement adéquat. On considérera comme comportement adéquat toute solution qui ne viole pas les contraintes données et qui ne résulte pas en un crash du programme.

Instructions :

1. Cet exercice doit être réalisé **individuellement !** L'échange de code relatif à cet exercice noté est **strictement interdit !** Cela inclut la diffusion de code sur le forum ou sur tout site de partage.

   Le code rendu doit être le résultat de *votre propre production*. Le plagiat de code, de quelque façon que de soit et quelle qu'en soit la source sera considéré comme de la tricherie (et peut même, suivant le cas, être illégal et passible de poursuites pénales).

   En cas de tricherie, vous recevrez la note « NA » pour l'entièreté de la branche et serez de plus dénoncés et punis suivant l'ordonnance sur la

discipline.

2. Vous devez donc également garder le code produit pour cet exercice dans un endroit à l'accès strictement personnel.

   Le fichier (source !) `mastermind.c` à fournir pour cet exercice ne devra plus être modifié après la date et heure limite de rendu.

3. Veillez à rendre du code *anonyme* : **pas** de nom, ni de numéro SCIPER !

4. Utilisez le site Moodle du cours pour rendre votre exercice.

   Vous avez jusqu'au mercredi 11 avril, 23:59 (heure du site Moodle du cours faisant foi) pour soumettre votre rendu.
   Aucun délai supplémentaire ne sera accordé.

Je tiens également à rappeler ici (dit dans le premier cours) que l'objectif de ce devoir n'est pas tant de vous évaluer que de vous entraîner. C'est pour cela que le coefficient de ce devoir est assez faible (1 sur 10). C'est également dans cet esprit qu'il a été conçu : de façon très progressive (de plus en plus difficile). Utilisez le donc comme un entraînement indicatif (sur lequel vous aurez un retour) en travaillant à votre niveau dans une durée qui vous semble raisonnable pour les crédits de ce cours (au lieu d'essayer de tout faire dans un temps déraisonnable).

II. Implémentation des IA pour le Mastermind

II.1. Cadre général

Le jeu Mastermind est un jeu de devinette entre deux joueurs. L'un d'entre eux sélectionne une combinaison de couleurs (4 couleurs parmi 6 dans la version standard) et l'autre doit la deviner sur la base de réponses à ses propositions. Les réponses indiquent à chaque proposition combien de couleurs sont correctement placées et combien d'autres couleurs proposées sont présentes dans la solution à deviner mais ne sont pas à la bonne place.

Dans ce devoir, votre programme jouera le rôle du joueur devant deviner la combinaison et l'utilisateur du programme jouera le rôle du joueur proposant une combinaison à deviner et répondant à chacune des propositions du programme.

Les ressources fournies pour ce devoir (à récupérer sur le site Moodle du cours) sont :

- un fichier `mastermind.c` contenant les bases du programme demandé ainsi qu'un certain nombre de « fonctions outils » fournies.

Les tâches à réaliser sont (nous vous conseillons de les réaliser dans cet ordre) :

1. définir les éléments généraux (types, fonctions principales) nécessaires au programme dans son ensemble ;
2. implémenter l'approche « *brute-force* » de l'IA ;
3. implémenter l'approche avec élimination des impossibles ;
4. implémenter l'approche de D. Knuth.

Pour réaliser ce devoir, vous pourriez être tentés de suivre l'une des deux stratégies extrêmes suivantes :

1. soit commencer par regarder le programme fourni et le remplir de sorte qu'il compile, en ne s'appuyant que sur les messages du compilateur sans lire la donnée ; puis ensuite implémenter la donnée pas à pas ;
2. implémenter la donnée pas à pas jusqu'au bout, puis compiler et enfin tester le tout.

La bonne façon de procéder nous semble cependant être d'alterner entre ces deux extrêmes :

1. au choix : commencez par compiler le code fourni ou lire *entièrement* la donnée ;
2. puis faites l'autre ;
3. une fois la donnée *entièrement* lue **et** le code d'origine essayé d'être compilé une première fois, essayez d'aller au plus vite vers un code *minimal* qui compile : définissez de façon minimale les types qui manquent : `BitSet`, `Combination` et `enum color` ; le code devrait alors compiler ;
4. procédez ensuite étape par étape suivant la donnée ; mais en **compilant** et **testant** votre code à chaque fois.

NOTE : il est clair que la présente donnée ne spécifie pas tout exhaustivement et que le code fourni en est un complément nécessaire permettant de préciser certains aspects de cette donnée !

II.2. Mise en place des éléments généraux

II.2.1 Combinaisons

Nous allons commencer ici par mettre en place quelques outils pour représenter les combinaisons de couleurs dans le jeu Mastermind. Définissez tout d'abord l'`enum color` qui peut prendre les valeurs `YELLOW`, `BLUE`, `GREEN`, `RED`, `PINK` ou `ORANGE` (définissez les dans cet ordre, ce sera important pour nos tests).

Ensuite, définissez le type `Combination` qui représente une combinaison de N couleurs. Par exemple, pour quatre couleurs, une telle combinaison pourrait être « `BLUE`, `ORANGE`, `RED`, `YELLOW` ».

Ce type `Combination` doit contenir la taille N de la combinaison concernée, ainsi qu'un tableau de couleurs *dynamiquement alloué* à la taille correspondante. Afin de facilement créer des instances de `Combination`, implémentez la fonction

`Combination create_combination(size_t)`

qui crée une combinaison de taille donnée en lui allouant la mémoire nécessaire pour son tableau dynamique et affectant toutes les couleurs à `YELLOW`.

Afin de permettre l'itération à travers les combinaisons, définissez la fonction `next_combination()` qui transforme une combinaison reçue en paramètre en une nouvelle « prochaine » combinaison suivant l'ordre des couleurs et en itérant en premier sur la première couleur.

Par exemple si la combinaison reçue est « `BLUE`, `ORANGE`, `RED`, `YELLOW` », la prochaine combinaison doit être « `GREEN`, `ORANGE`, `RED`, `YELLOW` » car `GREEN` est la couleur suivante après `BLUE` dans l'ordre donné plus haut.

Autre exemple : si la combinaison reçue est « `ORANGE`, `ORANGE`, `ORANGE`, `BLUE` », la prochaine combinaison doit être « `YELLOW`, `YELLOW`, `YELLOW`, `GREEN` », et la suivante sera « `BLUE`, `YELLOW`, `YELLOW`, `GREEN` », et ainsi de suite jusque « `ORANGE`, `YELLOW`, `YELLOW`, `GREEN` », puis « `YELLOW`, `BLUE`, `YELLOW`, `GREEN` », etc.

Si aucune prochaine combinaison n'existe, la fonction `next_combination()` retourne 0, sinon elle retourne 1.

Vous devez vous assurer qu'à partir d'une combinaison initiale obtenue par `create_combination()`, des appels successifs à `next_combination(proposition)` feront prendre à `proposition` toutes les valeurs de combinaisons possibles. En d'autres termes, une boucle du genre « `while (next_combination(proposition))` » permet de visiter toutes les combinaisons possibles.

II.2.2 Interaction avec l'utilisateur

Mettons maintenant en place l'interaction avec l'utilisateur. L'IA devra successivement proposer des combinaisons à l'utilisateur, lequel leur assignera un score en fonction de la combinaison secrète à laquelle il pense. Ses réponses doivent, bien entendu, rester cohérentes (il ne change pas de combinaison secrète en cours de route) sinon le comportement est indéterminé (nous ne testerons pas ces cas là).

Le code pour afficher une combinaison proposée vous est fourni dans la fonction `print_combination()`.

Afin de pouvoir lire la réponse de l'utilisateur, commencez par définir la structure `Answer` qui contient deux entiers non-signés `positions` et `colors`. La valeur de `positions` correspond au nombre d'éléments dans la combinaison proposée qui ont la même couleur **à la même position** que dans la combinaison de l'utilisateur. La valeur de `colors` correspond au nombre d'éléments dans la combinaison proposée qui ont la même couleur qu'un élément **mais à une position différente** dans la combinaison de l'utilisateur, et qui n'ont pas déjà été comptées dans `positions`.

Par exemple, si la combinaison secrète de l'utilisateur est « `ORANGE`, `ORANGE`, `GREEN`, `RED` » et que l'IA propose « `BLUE`, `ORANGE`, `RED`, `YELLOW` », la réponse de l'utilisateur devra être : 1 `positions` (pour le deuxième `ORANGE`) et 1 `colors` (pour le `RED`).

Pour l'interaction avec l'utilisateur, implémentez la fonction `ask()` qui reçoit une combinaison (premier argument) et une réponse (`Answer`) à remplir (en second argument) et :

1. affiche la combinaison reçue ;
2. affiche l'invitation à répondre « `Please score attempt (positions, colors):` » ;
3. lit ensuite la réponse de l'utilisateur et la stocke dans l'`Answer` reçue ;
   si l'utilisateur ne fourni pas deux entiers positifs, la fonction `ask()` affiche

   ```
   Unable to parse answer. Aborting.
   ```

   et retourne 1
   (on **ne** vous demande **pas** d'autres tests d'intégrité comme p.ex. `positions + colors` plus petit que la taille des combinaisons) ;
4. si l'utilisateur a marqué la solution comme gagnante (`positions` est égal à la taille de la combinaison (et `colors` est forcément à 0)), la fonction `ask()` affiche alors « `Found solution:` » (sans retour à la ligne) suivi de la combinaison gagnante, puis retourne 1.
   Sinon, la fonction retourne 0.

Notez que le code de retour de la fonction `ask()` représente donc en fait s'il faut s'arrêter de jouer (1) ou s'il faut continuer à chercher (0).

II.3. Algorithme « *brute-force* »

Sur la base des éléments développés jusqu'ici, vous devriez pouvoir implémenter la stratégie la plus naïve (« *brute-force* ») dans la fonction `solve_brute_force()` dont le prototype est fourni.

Le paramètre `size` de cette fonction indique la taille des combinaisons que l'IA doit trouver (la taille du jeu utilisé, 4 dans la version historique du Mastermind ; mais nous testerons aussi avec 2, 3 et 5).

Cette stratégie naïve itère simplement à travers toutes les combinaisons possibles et les présente successivement à l'utilisateur comme tentative. Si l'utilisateur marque la tentative comme bonne, l'itération termine comme indiqué ci-dessus.

Si l'itération prend fin sans qu'aucune solution n'ait été trouvée ou que l'utilisateur entre des données invalides, le programme termine immédiatement.

Exemple de déroulement :

```
What size (2, 3, 4, 5)? 4
What strategy ([B]rute force, B[i]tfield, [K]nuth)? B
Y Y Y Y
Please score attempt (positions, colors): 0 0
B Y Y Y
Please score attempt (positions, colors): 0 0
G Y Y Y
Please score attempt (positions, colors): 1 0
R Y Y Y
Please score attempt (positions, colors): 0 1
P Y Y Y
Please score attempt (positions, colors): 0 0
O Y Y Y
Please score attempt (positions, colors): 0 1
Y B Y Y
Please score attempt (positions, colors): 0 0
B B Y Y
[...]
^C     # (car l'utilisateur en a marre...)
```

II.3. Algorithme « *avec élimination* »

L'algorithme précédent ne prend pas du tout en compte les réponses de l'utilisateur (autre que la réponse « c'est tout bon ») et propose une à une des combinaisons même s'il elles sont incompatibles avec les réponses précédentes de l'utilisateur. Nous allons changer ce point en marquant certaines combinaisons comme impossibles (sachant les réponses de l'utilisateur).

Afin de marquer certaines combinaisons comme éliminées, nous allons implémenter un tableau de bits, une structure de données qui permet de marquer des index comme vrais ou faux en utilisant peu d'espace mémoire. Définissez pour cela le type `BitSet` qui contient un tableau de `char` (donc des éléments de 8 bits chacun) ainsi que la taille totale (en bits) du tableau de bits (= le nombre d'index qu'il peut marquer comme vrai ou faux). Les fonctions utilitaires pour créer le tableau de bits et le manipuler vous sont déjà fournies, assurez vous que votre définition du type soit compatible avec celles-ci.

Afin de permettre l'utilisation du tableau de bits pour marquer des combinaisons comme éliminées ou non, nous allons implémenter une bijection entre les instances de `Combination` (d'une taille N donnée) et les entiers dans l'intervalle $[0, 6^N[$ (nous avons 6 couleurs et donc 6^N combinaisons possibles). Cette bijection sera simplement la numérotation 6-aire des combinaisons en prenant les couleurs comme « chiffres » écrits dans l'ordre inverse de lecture. Par exemple, la combinaison « `BLUE`, `ORANGE`, `RED`, `YELLOW` » correspondra à l'index 139, puisque, en suivant l'ordre des couleurs donné plus haut, `YELLOW` vaut 0, `BLUE` vaut 1, `RED` 3 et `ORANGE` 5 (**Note :** il est garantit par la norme du C que les valeurs d'un

`enum` vont de 1 en 1 à partir de 0.), et donc « `BLUE`, `ORANGE`, `RED`, `YELLOW` » = 0351 en 6-aire, dans l'ordre inverse de lecture ; soit $1+5\times6+3\times6^2+0\times6^3 = 139$. (**Note :** le fait d'inverser l'ordre de lecture est en fait pour rendre le code *plus simple*, pas pour le rendre plus compliqué !! Les couleurs d'une combinaison sont alors en effet dans l'ordre croissant des puissances de 6 : la première couleur pour 6^0, la seconde pour 6^1, etc.). Notez aussi que l'ordre induit par cette bijection est consistant avec celui défini plus haut avec `next_combination()`, une propriété qui peut être utile à tester !

Définissez les fonctions :

1. `size_t combination_to_index(const Combination)` qui retourne l'index correspondant à une combinaison fournie en argument ;
2. `Combination* combination_from_index(size_t, Combination*)` qui transforme la combinaison fournie de façon à ce qu'elle corresponde à l'index fourni, et retourne ensuite cette combinaison (ceci est utile pour pouvoir passer des appels à `combination_from_index(...)` directement en argument d'autres fonctions).

Faîtes bien attention (= testez) à ce que les fonctions `combination_to_index()` et `combination_from_index()` soient bien des fonctions réciproques/inverses l'une de l'autre.

Il faut maintenant déterminer quand une combinaison peut être éliminée. Lorsque l'IA propose une combinaison tentative C_t et que l'utilisateur lui donne un score s_t=(`positions`, `colors`), ce score a été calculé par rapport à la combinaison C_{sol} secrète de l'utilisateur. Étant donnée la fonction $score(C_1, C_2)$ qui calcule le score (`positions`, `colors`) de la combinaison C_1 par rapport à la solution C_2, on a alors bien sûr $s_t = score(C_t, C_{sol})$, et l'ensemble $\mathcal{C} = \{C \mid score(C_t, C) = s_t\}$ est l'ensemble des combinaisons secrètes possibles pour ce score. L'intersection successive de ces ensembles au fur et à mesure que l'on reçoit des scores de la part de l'utilisateur constitue l'ensemble des combinaisons encore valables. C'est cette stratégie que nous vous demandons d'implémenter ici.

Pour cela, nous allons utiliser une structure `Solver_support` (à définir) utile pour maintenir les états intermédiaires dans la stratégie de recherche de nouvelles combinaisons. Cette structure doit contenir un tableau de bits ainsi que deux combinaisons. La première combinaison représente la tentative courante durant la procédure d'élimination, et la deuxième est une combinaison temporaire qui peut vous être utile, p.ex. comme argument à `combination_from_index()`.

Créez également toutes les fonctions outils que vous jugez nécessaires pour `Solver_support` (p.ex. création, destruction).

Une fois cette structure `Solver_support` définie, vous pouvez implémenter la procédure d'élimination avec le prototype suivant :

```
int review_combinations(Solver_support* s, size_t* count)
```

Cette fonction propose la tentative courante de `s` à l'utilisateur et reçoit le score assigné (fonction `ask()`).

Si la combinaison proposée était gagnante ou si l'utilisateur a entré des données invalides, la fonction retourne 0.

Sinon, elle marque comme éliminés tous les index du tableau de bits de `s` dont le score ne correspond pas à la réponse reçue. Pour cela, nous vous fournissons la fonction

```
int score_attempt(const Combination* attempt,
                  const Combination* result,
                  Answer* ans);
```

qui calcule le score (`positions`, `colors`) d'une combinaison proposée `attempt` par rapport à une combinaison supposée à deviner `result` et stocke ce score dans `ans` (lequel sera donc à comparer avec le score reçu de la part de l'utilisateur).

Concernant l'argument `count` de `review_combinations()` : si le pointeur reçu est valide, décrémentez la valeur pointée de 1 à chaque fois qu'un index est marqué comme éliminé. Cet argument ne nous sera pas utile pour la stratégie courante (passez `NULL` lors de son appel), mais le sera pour la stratégie suivante, l'algorithme de D. Knuth.

Au final, la fonction `review_combinations()` retourne 1 (si elle n'a pas retourné 0 pour les raisons invoquées plus haut, bien sûr !).

A l'aide des fonctions précédentes (et celles fournies pour les `BitSet`), implémentez l'approche par élimination dans la fonction

```
void solve_with_bitset(size_t size);
```

où `size` indique la taille des combinaisons que l'IA doit trouver (la taille du jeu).

Cette fonction commence par créer et initialiser un `Solver_support`, puis propose ensuite successivement des combinaisons qui n'ont pas encore été éliminées jusqu'à ce qu'une solution soit trouvée (ou que toutes les combinaisons aient été éliminées). Faîtes bien attention à proposer à chaque itération la **prochaine** combinaison qui n'a pas encore été éliminée selon l'ordre défini par `index_to_combination()` (en commençant par « `YELLOW`, `YELLOW`, `YELLOW`, `YELLOW` » = index 0). Cette propriété sera importante pour nos tests. Lorsqu'une solution gagnante est trouvée par l'IA, elle affiche le message tel qu'indiqué dans la fonction `ask()` ; et si toutes les combinaisons ont été éliminées ou que l'utilisateur entre des données invalides (retour 1 de la fonction `ask()`), le programme termine immédiatement.

Indication : avec la modularisation proposée (fonctions précédentes), cette fonction `solve_with_bitset()` est très courte à écrire, moins de 10 lignes.

Exemple de déroulement pour la combinaison secrète « `ORANGE`, `GREEN`, `GREEN`, `RED` »

```
What size (2, 3, 4, 5)? 4
What strategy ([B]rute force, B[i]tfield, [K]nuth)? i
Y Y Y Y
Please score attempt (positions, colors): 0 0
B B B B
Please score attempt (positions, colors): 0 0
G G G G
Please score attempt (positions, colors): 2 0
R R G G
Please score attempt (positions, colors): 1 2
P G R G
Please score attempt (positions, colors): 1 2
O G G R
Please score attempt (positions, colors): 4 0
Found solution: O G G R
```

II.4. Algorithme de Donald Knuth.

Notes préliminaires :

1. cette partie du devoir est la plus « *challenging* » (et nécessite donc une certaine réflexion de votre part) ;
2. n'hésitez pas consulter la documentation référencée (hyperliens) si nos explications ne vous semblent pas assez claires.

[fin de notes]

L'approche de Donald Knuth améliore l'approche par élimination précédente en sélectionnant pour chaque nouvelle tentative la combinaison qui éliminera le maximum de combinaisons restantes. Cette nouvelle tentative est obtenue par une approche minimax qui sélectionne la combinaison qui maximise le nombre minimum de combinaisons éliminées.

Pour trouver cette nouvelle combinaison à proposer, on commence par représenter l'ensemble R des combinaisons restantes à ce stade (fonction `review_combinations()`, dont il pourra être utile ici d'utiliser le paramètre `count`). On itère ensuite sur toutes les combinaisons (y compris les éliminées ! donc pas sur R ici, mais bien sur l'ensemble complet de toutes les combinaisons existantes pour le jeu au départ). Appelons C_i la combinaison courante de cette itération. Le but est alors de déterminer le nombre *minimum* de combinaisons restantes qui seront éliminées si l'IA choisit C_i comme tentative.

Pour chaque score s=(`positions`, `colors`), on calcule la cardinalité de l'ensemble $H_i(s) = \{C \in R \mid score(C_i, C) = s\}$; $H_i(s)$ correspond donc à l'ensemble des combinaisons solutions (non éliminées) qui donneraient le score s pour la tentative C_i.

Une propriété importante de $H_i(s)$ est que sa cardinalité correspond au nombre de combinaisons qui resteront dans R si la tentative C_i est choisie et le score s est assigné par l'utilisateur. Ainsi, on obtient le nombre *minimum* de combinaisons éliminées en prenant la différence entre la cardinalité de R (argument `count` de la fonction `review_combinations()`) et le maximum des cardinalités des différents $H_i(s)$ (i fixé, s variable).

Indications :

1. Pour calculer les cardinalités des différents $H_i(s)$ (i fixé, s variable), vous pouvez utiliser un tableau T d'entiers indexé par chaque score s=(`positions`, `colors`) possible. Itérez sur les $C \in R$ et pour chaque score s obtenu par $s = score(C_i, C)$, incrémentez la valeur à l'index s dans votre tableau T. La cardinalité maximale est ensuite obtenue en trouvant la valeur maximale dans le tableau T.
2. La cardinalité de R vaut initialement le nombre total de combinaisons possibles (argument `count` de la fonction `review_combinations()`), puis chaque fois qu'une combinaison est éliminée (en d'autres termes, retirée de R), cette valeur est décrémentée de 1.
3. N'hésitez pas à **modulariser** votre code !

[**fin des indications**]

Votre but est d'éliminer le maximum de combinaisons avec votre tentative. Vous devrez donc choisir la tentative pour laquelle le nombre minimum de combinaisons éliminées est maximal. En cas d'égalité, assurez vous d'abord de choisir une combinaison qui n'a pas encore été éliminée parmi les différentes combinaisons avec un score maximal (si possible), et ensuite de choisir la combinaison avec l'index (selon `combination_to_index()`) le plus petit. La première propriété est nécessaire pour trouver la solution en 5 tentatives ou moins, tandis que la deuxième sera importante pour nos tests.

Implémentez la stratégie de Donald Knuth dans la fonction

```
void solve_knuth(size_t size);
```

où `size` indique la taille des combinaisons que l'IA doit trouver.

La fonction commence par créer et initialiser un `Solver_support` et met la première moitié des couleurs à `YELLOW` et la deuxième moitié à `BLUE`. Si `size` est impair, on aura un `YELLOW` de plus que les `BLUE`.

La stratégie propose ensuite la combinaison initiale puis sélectionne à chaque itération la combinaison qui maximise le nombre de combinaisons potentiellement éliminées jusqu'à ce qu'une solution soit trouvée (ou que toutes les combinaisons soient éliminées). Utilisez la fonction `ask()` implémentée précédemment pour communiquer avec l'utilisateur et faites bien attention d'utiliser les autres fonctions définies précédemment à chaque fois que possible. Lorsqu'une solution gagnante est trouvée par l'IA, elle affiche le message tel qu'indiqué dans la fonction `ask()` ; et si toutes les combinaisons ont été éliminées ou que l'utilisateur

entre des données invalides (retour 1 de la fonction `ask()`), le programme termine immédiatement.

Exemple de déroulement pour la combinaison secrète « `ORANGE`, `GREEN`, `GREEN`, `RED` »

```
What size (2, 3, 4, 5)? 4
What strategy ([B]rute force, B[i]tfield, [K]nuth)? K
Y Y B B
Please score attempt (positions, colors): 0 0
P R G G
Please score attempt (positions, colors): 1 2
R P R G
Please score attempt (positions, colors): 0 2
O G R Y
Please score attempt (positions, colors): 2 1
O G G R
Please score attempt (positions, colors): 4 0
Found solution: O G G R
```

III. Conseils et tests

N'hésitez pas à créer d'autres fonctions utilitaires si nécessaire !

Tout votre code et toutes vos fonctions doivent être robustes tant que faire se peut (c.-à-d. sauf avis contraire).

Pensez à tester correctement chacune des fonctionnalités implémentées **avant** de passer à l'étape suivante. Cela vous évitera bien des tracas…

Note : pour vous faciliter vos tests, vous pouvez créer l'entrée désirée dans un fichier texte et le donner en entrée standard sur un terminal. Par exemple, vous pouvez créer le fichier `OGGR_in.txt` contenant

```
4 K
0 0
1 2
0 2
2 1
4 0
```

et lancer

```
./mastermind < OGGR_in.txt
```

Cela vous évitera de toujours avoir à tout retaper à chaque fois.