Master-mind(tm) [tb06] - Exercice

Karine Zampieri, Bruno Adam, Stéphane Rivière

Unisciel algoprog Université Haute-Alsace

Version 19 mai 2018

Table des matières

Python - Master-mind(tm) (Solution)

Mots-Clés Tableau unidimensionnel, Jeu ■
Requis Structures de base, Structures conditionnelles, Algorithmes paramétrés, Structures répétitives, Schéma itératif ■
Difficulté ● ● ○

Objectif
Cet exercice réalise un jeu de Master-Mind(tm).

...(énoncé page suivante)...

1 Énoncé

Le Master-Mind(tm)
C'est un **jeu de logique** qui se joue à deux. Généralement, il se présente sous la forme d'un plateau ayant 12 rangées de 4 trous pouvant accueillir des pions de couleurs. Le nombre de couleurs est 8 et sont : rouge, jaune, vert, bleu, orange, noir, marron, fuchsia. (image : https://fr.wikipedia.org/wiki/Mastermind)

Des fiches blanches et noires (ou rouges selon les versions du jeu, cf. figure ci-avant, sur la droite) seront utilisées pour donner des indications à chaque étape du jeu.

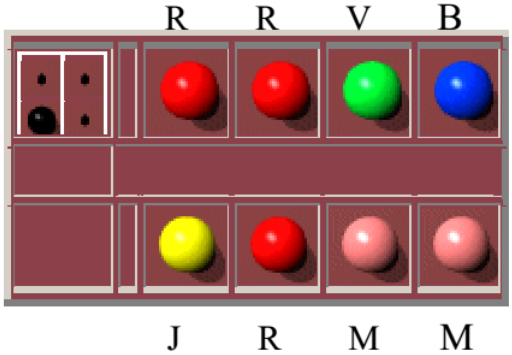
Il existe de nombreuses variantes selon le nombre de couleurs, de rangées ou de trous.

Règles du jeu
Avant de démarrer la partie, le second joueur place ses pions de couleurs derrière le cache (voir figure) pour éviter que le joueur ne voie les pions choisis. Le but est de retrouver quels sont les pions choisis par l'autre joueur et d'en connaître les positions.

Pour cela, à chaque tour, le joueur doit se servir des pions de couleurs pour former une rangée. Une fois les pions placés, l'autre joueur indique le nombre de pions de bonne couleur :

1. **Bien placés** en utilisant les fiches noires.
2. **Mal placés** avec les fiches blanches.

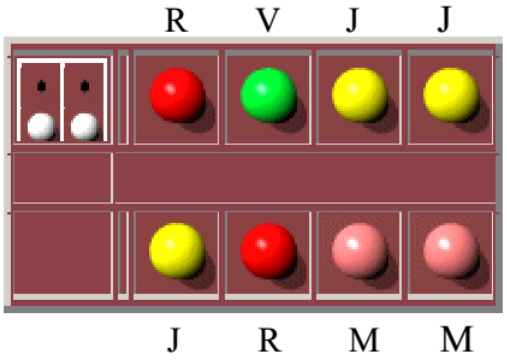
Exemple
La ligne du haut est celle du joueur et la ligne du bas est la combinaison à trouver :



La fiche noire correspond au deuxième pion Rouge bien placé. On n'obtient rien pour le premier pion Rouge car la combinaison ne contient pas deux fois la couleur Rouge.

Exemple

Ici le premier pion Rouge et le troisième pion Jaune sont mal placés.



Fin d'une partie

Deux situations terminent une partie :

- Le joueur n'a pas réussi à trouver la solution : la partie est perdue.
- Le joueur a trouvé la solution : la partie est gagnée.

Objectif

Écrire une version programmée du jeu « Machine contre Humain ».

...(suite page suivante)...

2 Jeu Machine-contre-Humain

Le programme (= machine) tire une combinaison au hasard et demande au joueur (= humain) de la retrouver sur la base de ses réponses successives. Le programme indique à chaque coup **combien** d'éléments sont bien devinés et correctement placés (par `#`) ou bien devinés mais mal placés (par `o`). Le joueur dispose de `maxcoups` tentatives. Une combinaison est représentée par une séquence de n symboles tirés parmi m.

Voici un exemple du résultat attendu :

```
Pouvez-vous trouver ma combinaison de 4 symboles
[chiffres entre 1 et 6 avec repetitions possibles]
en moins de 10 coups? Entrez les symboles des
propositions terminees par [Entree].
(# un bien place, o un mal place)

Votre proposition? 1 2 3 4
#o (reste 9 coups)
Votre proposition? 2 5 6 3
oo (reste 8 coups)
Votre proposition? 5 4 2 1
#oo (reste 7 coups)
Votre proposition? 5 2 2 5
##o (reste 6 coups)
Votre proposition? 1 2 1 5
##o (reste 5 coups)
Votre proposition? 5 2 1 2
#### (reste 4 coups)
Bravo! Vous avez trouve en 6 coups
```

2.1 Représentation du jeu

Définissez la constante `CMAX=6` (nombre maximum de symboles) puis le type `Combinaison` comme étant un tableau d'entiers de taille maximale `CMAX`.

Écrivez une procédure `genererCmb(c,n,m)` qui tire (au hasard) une `Combinaison c` de `n` symboles dans `[1..m]`. Exemple : `genererCmb(c,4,6)` doit générer `4(=n)` entiers compris dans `[1..6(=m)]` dans `c`.

Outil Python

Le package `random` définit la fonction `randint(a,b)` qui renvoie un entier pseudo-aléatoire compris dans l'intervalle `[a..b]`.

Validez vos définitions et votre procédure avec la solution.

Solution Python @[pgtmind.py]

```
CMAX = 6
""" Taille maximale des Combinaison """
```

```
def genererCmb(c, n, m):
    """  Génère une Combinaison de n symboles dans [1..m]

     :param c: une Combinaison
     :param n: taille des Combinaison
     :param m: nombre de couleurs
    """
    for k in range(0, n):
        c[k] = rd.randint(1, m)
```

Écrivez une procédure `saisirCmb(c,n,m)` qui saisit une `Combinaison` dans `c` de `n` symboles dans `[1..m]`. Supposez que les symboles sont effectivement dans `[1..m]`. Affichez l'invite :

```
Votre proposition?
```

Afin de vérifier la génération ainsi que la saisie, écrivez une procédure `afficherCmb(c,n)` qui affiche une `Combinaison c` de `n` symboles.

Validez vos procédures avec la solution.

Solution Python @[pgtmind.py]

```
def saisirCmb0(c, n, m):
    """  Saisit la Combinaison du joueur

     :param c: une Combinaison
     :param n: taille de c
     :param m: nombre de couleurs
    """
    print(n, " chiffres dans [1..", m, "]? ", sep="", end="")
    for k in range(0, n):
        c[k] = int(input())
```

```
def afficherCmb0(c, n):
    """  Affiche une Combinaison

     :param c: une Combinaison
     :param n: taille de c
    """
    for k in range(0, n):
        print(c[k], " ", sep="", end="")
    print()
```

Écrivez une procédure `test_cmb` qui teste vos deux procédures ci-avant, à savoir :

- Générez une `Combinaison m` puis affichez-la.
- Saisissez une `Combinaison j` puis affichez-la.

Appelez vos procédures avec `n=4` et `m=6`.

Testez.

Validez votre procédure avec la solution.

Solution Python @[pgtmind.py]

```
def test_cmb():
    """ @test """
    mystere = [0 for x in range(CMAX)]
    genererCmb(mystere, 4, 6)
    afficherCmb0(mystere, 4)

    humain = [0 for x in range(CMAX)]
    saisirCmb0(humain, 4, 6)
    afficherCmb0(humain, 4)
```

2.2 Étude du code d'un humain

Une réponse est constituée d'indicateurs (au plus n) :

- Ceux correspondant aux **symboles corrects bien placés**, c.-à-d. les valeurs et positions exactement les mêmes entre les deux séquences. Exemple : Si la séquence à deviner est 1213 et que le joueur propose 4516, la réponse sera « 1 correct » ce qui correspond au 1. Ce type de réponse sera noté par un « # » et marqué de façon interne par la constante `BIEN_PLACE`.
- Ceux correspondant aux **symboles corrects mais mal placés**, c.-à-d. une valeur présente mais pas à la bonne position. Exemple : Si la séquence à deviner est 1213 et que le joueur propose 4562, la réponse sera « 1 mal placé » ce qui correspond au 2. Ce type de réponse sera noté par un « o » et marqué de façon interne par la constante `MAL_PLACE`.

Ces indicateurs sont cumulés pour tous les symboles de la réponse. Exemple : Si la séquence à deviner est 1213 et que le joueur propose 4512, la réponse sera « #o » ce qui correspond au 1 et au 2.

Rappel

Il y a deux principes dans la génération des réponses :

1. Un symbole ne peut servir qu'**une seule fois** dans la réponse. Exemple : Si la séquence à deviner est 1213 et que la proposition est 4156, la réponse est « o » (c.-à-d. un bon symbole mais mal placé : le 1) et non pas « oo » (pour le 1 deux fois mal placé) car il n'y a qu'**un seul** 1 dans la séquence proposée. Par contre, si la proposition est 4134, la réponse sera bien « oo » (le 1 et le 3).
2. On ne précise pas quels symboles sont bien ou mal placés. On indique uniquement leur nombre. Exemple : Si la séquence à deviner est 4213 et que la proposition est 5243, la réponse sera « ##o » et non pas « _#o# » (où _ désigne un blanc).

Définissez les constantes `BIEN_PLACE=-1` et `MAL_PLACE=0`.
Notez que ces deux valeurs ne correspondent à aucune couleur.

Écrivez une fonction `trouverBienPlaces(copie,c,n)` qui trouve et fixe les biens placés dans une `Combinaison copie` du code secret. La `Combinaison c` est celle du joueur. L'entier `n` désigne la taille des combinaisons. La fonction doit remplacer dans `copie` tout symbole bien placé par un `BIEN_PLACE` et **retourner** le nombre de biens placés (entier). Exemple :

```
Pour:  copie=[1, 4, 3, 2] avec n=4, m=6
           c=[2, 4, 1, 6]
Alors: copie=[1,-1, 3, 2] et nbp=1
```

Solution simple

Cela consiste en :

- Compter le nombre de couleurs bien placées, c.-à-d. comparer et compter un à un les éléments à la même position dans les deux tableaux.
- Pour éviter de confondre les couleurs bien placées et les mal placées, il faudra éliminer les couleurs déjà comptées. Pour ce, on la remplace par la valeur du `BIEN_PLACE`.

Validez vos définitions et votre fonction avec la solution.

Solution Python @[pgtmind.py]

```python
BIEN_PLACE = -1
""" Valeur du Bien-placé '#' """
MAL_PLACE = 0
""" Valeur du Mal-placé 'o' """
```

```python
def trouverBienPlaces(copie, c, n):
    """  Trouve et positionne les biens placés dans la copie

     :param copie: Combinaison du code secret
     :param c: Combinaison du joueur
     :param n: taille des Combinaison
     :return: le nombre de bien placés
    """
    nbp = 0
    for k in range(0, n):
        if copie[k] == c[k]:
            nbp += 1
            copie[k] = BIEN_PLACE
    return nbp
```

Écrivez une fonction `position(valeur,c,n)` qui recherche l'entier de valeur `valeur` dans les `n` symboles d'une `Combinaison c`. Dans le cas d'une recherche fructueuse, la fonction renvoie le premier indice `k` tel que `c[k]` vaut `valeur`, -1 sinon. Exemples :

```
Indices: 0  1  2  3
Pour: c=[1,-1, 3, 2] avec n=4, m=6
Alors: position(2,c,n) renvoie 3
   Et: position(6,c,n) renvoie -1
```

Déduisez une fonction `trouverMalPlaces(copie,c,n)` qui trouve et fixe les mal placés dans une `Combinaison copie` du code secret. La `Combinaison c` est celle du joueur. L'entier `n` désigne la taille des combinaisons. La fonction doit remplacer dans `copie` tout symbole deviné mal placé par un `MAL_PLACE` et **retourner** le nombre de mal placés (entier). Exemple :

```
Indices:      0  1  2  3
Pour:  copie=[1,-1, 3, 2] avec n=4, m=6
           c=[2, 4, 1, 6]
Alors: copie=[1,-1, 3, 0] apres position(c[0]=2,copie,n) qui renvoie 2
            =[0,-1, 3, 0] apres position(c[2]=1,copie,n) qui renvoie 0
            =[0,-1, 3, 0] apres position(c[3]=6,copie,n) qui renvoie -1
Et: nmp=2
```

On notera que `position(c[1]=4,copie,n)` n'a pas été appelé car le symbole était `BIEN_PLACE` dans `copie`.

Solution simple

Ici cela consiste en :

- Parcourir la `Combinaison c` du joueur, et pour chaque élément qui n'est pas bien placé (c.-à-d. dont la valeur est différente de `BIEN_PLACE` dans la `copie`), rechercher sa position dans la `copie`.
- Si cet élément existe, il faut le compter puis en modifier sa valeur par celle du `MAL_PLACE` dans la `copie` pour qu'il ne puisse être compté deux fois.

Validez vos fonctions avec la solution.

Solution Python @[pgtmind.py]

```python
def position(valeur, c, n):
    """  Recherche linéaire d'une couleur dans une Combinaison

     :param valeur: une couleur
     :param c: Combinaison du joueur
     :param n: taille de c
     :return: l'indice de valeur dans c, -1 si elle n'existe pas
    """
    k = 0
    while k < n and c[k] != valeur:
        k += 1
    return (k if k < n else -1)
```

```python
def trouverMalPlaces(copie, c, n):
    """  Trouve et positionne les mal placés dans la copie

     :param copie: Combinaison du code secret
     :param c: Combinaison du joueur
     :param n: taille des Combinaison
     :return: le nombre de mal placés
    """
    nmp = 0
    for k in range(0, n):
        if copie[k] != BIEN_PLACE:
            p = position(c[k], copie, n)
            if p != -1:
                nmp += 1
                copie[k] = MAL_PLACE
    return nmp
```

Modifiez votre procédure `afficherCmb(c,n)` qui affiche une `Combinaison c` de `n` symboles de sorte qu'elle affiche :

- Un `'#'` si le symbole est bien placé.
- Un `'o'` si le symbole est mal placé.
- Et le symbole sinon.

Exemple :

```
Pour: c=[0,-1, 3, 0] et n=4 ==> o#3o
```

Validez votre procédure avec la solution.

Solution Python @[pgtmind.py]

```python
def afficherCmb(c, n):
    """  Affiche une Combinaison

     :param c: une Combinaison
     :param n: taille de c
    """
    for k in range(0, n):
        valeur = c[k]
        if valeur == BIEN_PLACE:
            print("#", end="")
        elif valeur == MAL_PLACE:
            print("o", end="")
        else:
            print(valeur, end="")
    print()
```

Copiez/collez la procédure `test_cmb` en la procédure `test_code` puis **complétez**-la de sorte qu'elle :

1. Appelle `trouverBienPlaces` où le paramètre muet `copie` est la `Combinaison m` de la machine et le paramètre muet `c` est la `Combinaison j` du joueur.

2. Affiche `m` ainsi que le nombre de biens placés calculés.
3. Appelle `trouverMalPlaces` selon le même principe que (1).
4. Affiche `m` ainsi que le nombre de mal placés calculés.

Testez.

Validez votre procédure avec la solution.

Solution Python @[pgtmind.py]

```
def test_code():
    """ @test """
    mystere = [0 for x in range(CMAX)]
    genererCmb(mystere, 4, 6)
    afficherCmb(mystere, 4)

    humain = [0 for x in range(CMAX)]
    saisirCmb0(humain, 4, 6)
    afficherCmb(humain, 4)

    nbp = trouverBienPlaces(mystere, humain, 4)
    afficherCmb(mystere, 4)
    print("nbp = ", nbp, sep="")
    nmp = trouverMalPlaces(mystere, humain, 4)
    afficherCmb(mystere, 4)
    print("nmp = ", nmp, sep="")
```

Écrivez une procédure `copierCmb(src,dest,n)` qui recopie les `n` symboles d'une `Combinaison src` dans une `Combinaison dest`.

Finalement écrivez une procédure `etudierCode(c1,c2,n,nbp,nmp)` qui étudie les `n` (entier) symboles d'une `Combinaison c2` d'un humain par rapport aux `n` symboles d'une `Combinaison c1` d'une machine. La procédure **restitue** le nombre de symboles bien placés dans `nbp` (entier) et le nombre de symboles devinés mal placés dans `nmp` (entier).

Solution simple

Il faut :

- Copier la `Combinaison` de la machine `c1` dans une `copie`.
- Trouver d'abord les biens placés.
- Trouver ensuite les mal placés.

Validez vos procédures avec la solution.

Solution Python @[pgtmind.py]

```
def copierCmb(src, dest, n):
    """  Recopie d'une Combinaison

     :param src: Combinaison source
     :param dest: Combinaison destination
     :param n: taille des Combinaison
    """
    for k in range(0, n):
        dest[k] = src[k]
```

```
def etudierCode(c1, c2, n):
    """  Etudie le code d'un humain p/r à celui de la machine

     :param c1: Combinaison du code secret
     :param c2: Combinaison du joueur
     :param n: taille des Combinaison
     :return:
    """
    copierCmb(c1, copie, n)
    nbp = trouverBienPlaces(copie, c2, n)
    nmp = (trouverMalPlaces(copie, c2, n) if nbp != n else 0)
    return (nbp, nmp)
```

2.3 Mise en place du tout

Écrivez une procédure `mastermind(n,m,maxcoups)` qui lance une partie de Master-Mind(tm) où l'entier `n` désigne le nombre de cases, l'entier `m` le nombre de symboles et l'entier `maxcoups` le nombre de coups maximum. La procédure doit :

- Générer la combinaison mystère.
- Lancer la partie et itérer tant que le nombre de coups joués est plus petit que `maxcoups` et que le joueur n'a pas trouvé.
- Afficher le score (message de réussite ou d'échec) du joueur.

Écrivez un script qui lance une partie de Master-Mind(tm) avec $n = 4$ et $m = 6$ (valeurs standard) en fixant 10 pour le nombre maximum de coups.

Testez.

Validez votre procédure avec la solution.

Solution Python @[pgtmind.py]

```
def mastermind(n, m, maxcoups):
    """  Lance une partie du jeu de MasterMind

     :param n: taille des Combinaison
     :param m: nombre de couleurs
```

```
     :param maxcoups: nombre de coups maximum de l'humain
    """
    # solution à trouver
    mystere = [0 for x in range(CMAX)]
    genererCmb(mystere, n, m)
    # nombre de coups effectués
    ncoups = 0
    # proposition == solution ?
    trouve = False
    # Lance la partie
    while not trouve and (ncoups < maxcoups):
        # proposition de l'humain
        humain = [0 for x in range(CMAX)]
        saisirCmb(humain, n, m)
        # Un coup de plus
        ncoups += 1
        nbp, nmp = etudierCode(mystere, humain, n)
        trouve = (nbp == n)
        # Résultat du code
        for j in range(nbp):
            print("#", end="")
        for j in range(nmp):
            print("o", end="")
        print(s, " (reste ", (maxcoups - ncoups), " coups)", sep="")
    # Affiche le score de la partie
    if trouve:
        print("GAGNE en ", ncoups, " coups", sep="")
    else:
        print("PERDU... voici la combinaison: ", end="")
        afficherCmb(mystere, n)
```

2.4 Jeu avec niveaux

Pour que le jeu soit plus intéressant, le joueur pourra choisir son niveau de difficulté parmi : Novice, Pro, Killer.

Écrivez une procédure `saisirNiveau(n,m,maxcoups)` qui saisit le niveau puis détermine et **restitue** le nombre de symboles dans `n` (entier), le nombre de couleurs dans `m` (entier) et le nombre maximum de coups dans `maxcoups` (entier) comme suit :

- 4 couleurs parmi 8 au niveau Novice, 5 parmi 8 en Pro et 6 parmi 8 en mode Killer.
- Le nombre de rangées (nombre maximum de propositions) est limité à 12 en mode Novice, à 15 en mode Pro et à 20 en mode Killer.

Affichez l'invite :

```
Niveau N)ovice, P)ro ou K)iller?
```

Écrivez une procédure `bienvenue(n,m,maxcoup)` qui affiche un texte d'accueil présentant le jeu où l'entier `n` désigne le nombre de cases, l'entier `m` le nombre de symboles et l'entier `maxcoups` le nombre maximum de coups. Reportez-vous à l'exemple d'exécution (en début de la section 2, page 4) pour définir le texte de cette procédure. Libre à vous d'agrémenter le texte d'accueil comme bon vous inspire.

Validez vos procédures avec la solution.

Solution Python @[pgtmind.py]

```
def saisirNiveau():
    """  Saisit le niveau du jeu

     :return: le tuple (n,m,maxcoup)
    """
    n, m, maxcoups = 0, 0, 0
    c = "."
    while not (c == "n" or c == "p" or c == "k"):
        c = input("Niveau n)ovice, p)ro ou k)iller? ")[0]
    if c == "n":
        n, m, maxcoups = 4, 8, 12
    elif c == "p":
        n, m, maxcoups = 5, 8, 15
    elif c == "k":
        n, m, maxcoups = 6, 8, 20
    return (n, m, maxcoups)
```

```
def bienvenue(n, m, maxcoups):
    """  Affiche un texte d'accueil présentant le jeu

     :param n: taille des Combinaison
     :param m: nombre de couleurs
     :param maxcoups: nombre de coups maximum de l'humain
    """
    print("Pouvez-vous trouver ma combinaison de ", n, " symboles", sep="")
    print("[chiffres entre 1 et ", m, " avec repetitions possibles]", sep="")
    print("en moins de ", maxcoups, " coups? Entrez les symboles des", sep="")
    print("propositions terminees par [Entree].")
    print("(# un bien place, o un mal place)", '\n', sep="")
```

Avant votre procédure `saisirCmb`, écrivez une fonction `cmbValide(c,n,m)` qui teste et renvoie `Vrai` si chaque symbole d'une `Combinaison c` de `n` symboles est compris dans `[1..m]`, `Faux` sinon.

Modifiez votre procédure `saisirCmb(c,n,m)` de sorte qu'elle saisit une `Combinaison` dans `c` de `n` symboles dans `[1..m]` **tant qu'elle n'est pas valide**.

Validez votre fonction et procédure avec la solution.

Solution Python @[pgtmind.py]

```
def cmbValide(c, n, m):
    """  Prédicat de Combinaison valide

     :param c: une Combinaison
     :param n: taille de c
     :param m: nombre de couleurs
```

```
     :return: Vrai si c est valide (n symboles dans [1..m])
    """
    ok = True
    k = 0
    while k < n and ok:
        ok = ((1 <= c[k]) and (c[k] <= m))
        k += 1
    return ok
```

```
def saisirCmb(c, n, m):
    """  Saisie contrainte de la Combinaison du joueur

     :param c: une Combinaison
     :param n: taille de c
     :param m: nombre de couleurs
    """
    saisirCmb0(c, n, m)
    while not cmbValide(c, n, m):
        saisirCmb0(c, n, m)
```

Écrivez une procédure `mastermind2` qui :

- Demande le niveau du jeu.
- Affiche le principe du jeu.
- Appelle la procédure `mastermind` avec les bons paramètres.

Testez.

Validez votre procédure avec la solution.

Solution Python @[pgtmind.py]

```
def mastermind2():
    """ Jeu du mastermind """
    n, m, maxcoups = saisirNiveau()
    bienvenue(n, m, maxcoups)
    mastermind(n, m, maxcoups)
```

2.5 Validation

Validez votre script avec la solution.

3 Références générales

Comprend [] ■