Mastermind : Des preuves par ordinateur

Bernard Langer(*)

Mastermind est un jeu à deux joueurs apparu en 1973 qui a connu un vif succès à sa sortie. Il a été décliné en plusieurs variantes augmentant sa difficulté initiale. Ce jeu a fait l'objet de diverses études mathématiques. Citons en particulier celle de Donald Knuth (pionnier de l'algorithmique, à l'origine du langage *TeX*) parue en 1977.

1. Introduction : Présentation du jeu.

Cette photo illustre une partie de Mastermind où le décodeur a trouvé le code secret en 6 étapes. Les boules ont été numérotées pour éviter les confusions inévitables dans une impression en bichromie.

Sous sa forme classique, le jeu de Mastermind est constitué d'un plateau perforé de 12 rangées de 4 trous dans lesquels seront disposées des billes de couleurs. Chaque rangée est complétée par 4 trous plus petits pouvant accueillir des *fiches* blanches ou rouges (les trous situés sur le côté gauche du plateau ci-contre ne servent qu'à mémoriser les vainqueurs des diverses parties)

Deux joueurs s'affrontent : le *codeur* et le *décodeur*.

- Le codeur choisit un « code secret » constitué d'un alignement de 4 billes colorées, les couleurs pouvant être répétées. Ce code n'est pas visible par le décodeur dont le rôle consiste à le reconstituer.
- Le décodeur propose un code en plaçant 4 billes dans l'une des rangées du plateau. Le codeur répond à cette proposition en plaçant autant de fiches rouges que de *billes de la bonne couleur et bien placées* puis autant de fiches blanches que de *billes de la bonne couleur mais mal placées*.

(*) bernard.langer@laposte.net

Par exemple :

 Code secret	 Proposition du décodeur
Dans la proposition du décodeur, seule la deuxième bille (en partant de la gauche) est bien placée. Le codeur place une fiche rouge sur le plateau. La quatrième bille du décodeur a la même couleur (verte ou 1) que la première du code secret. Le codeur place une fiche blanche sur le plateau. Une seconde fiche blanche correspondant aux billes rouges est également placée	

Lorsqu'on utilise le plateau illustré ci-dessus, le décodeur gagne la partie s'il finit par obtenir 4 fiches rouges en 12 tentatives ou moins ; le codeur gagne si le décodeur n'a pas trouvé au bout de 12 tentatives.
Pour éviter toute ambiguïté nous supposerons que :

- Tous les trous contiennent une bille. Ce n'est pas une véritable restriction puisqu'il suffirait de convenir que le vide est une couleur supplémentaire.
- L'évaluation d'une proposition se fait dans l'ordre : fiches rouges *puis* fiches blanches en ne tenant plus compte des occurrences déjà décomptées.

La forme classique du jeu comporte 4 trous et autorise 6 couleurs différentes.

En utilisant les ressources de l'informatique nous dégagerons une stratégie gagnante côté décodeur sans cependant avoir la garantie que cette stratégie soit optimale mais en *démontrant* qu'elle conduit toujours au succès.
Cet article peut se lire sans nécessairement se plonger dans le codage des algorithmes, néanmoins nous invitons le lecteur intéressé à télécharger les programmes correspondants sur le site de l'APMEP.
Les algorithmes étudiés sont courts mais non triviaux. Certains d'entre-eux se prêtent à des activités dans le cadre de l'ISN.

Nous essaierons de respecter les grands principes de la démarche informatique :

- Analyse du problème.
- Construction de l'algorithme.
- (Codage dans un langage de programmation)
- Exploitation des résultats.

Pour coder les algorithmes, nous avons retenu le langage Python, moderne et très flexible. L'avatar *Amiens-Python* développé par les collègues de l'Académie d'Amiens s'est montré particulièrement adapté à nos objectifs.
Cette version est librement téléchargeable à l'adresse :

http://amienspython.tuxfamily.org/

2. Analyse de la situation

Si l'utilisation des billes colorées ainsi que des fiches rouges et blanches convient parfaitement au jeu humain, il n'en va pas de même avec l'ordinateur. On pourrait bien sûr programmer la machine pour qu'elle imite parfaitement le jeu de plateau mais puisque notre objectif est d'analyser le jeu d'un point de vue mathématique nous allons utiliser un codage numérique dont les avantages vont apparaître immédiatement.

- En codant les couleurs par un nombre choisi dans C = {0,1,2,3,4,5} nous pourrons assimiler un code à une 4-liste d'éléments de C. Il existe une façon naturelle d'engendrer toutes ces listes : il suffit pour cela de compter de 0 à 5 555… en base 6. Il faudra cependant compléter les nombres qui s'écrivent avec moins de 4 chiffres par des « 0 » avant le premier chiffre non nul.
  Nous utiliserons donc deux fonctions (analysées en annexe) :
  – *Base*(*n*,*b*) qui transforme l'entier *n* en base *b* (*n* et *b* étant écrits en base dix). Ainsi *Base*(51,6) = 123.
  – *Proposition*(*n*) qui construit une 4-liste en isolant les chiffres de *n*. Par exemple *Proposition*(123)=[0 ,1, 2, 3].
- Puisque le nombre de fiches rouges et blanches est nécessairement compris entre 0 et 4, nous pourrons représenter la réponse du décodeur par un entier de deux chiffres, le chiffre des dizaines étant égal au nombre de fiches rouges et le chiffre des unités à celui des fiches blanches. Nous appellerons ce nombre score de la proposition. Les 14 scores possibles sont :
  0, 1, 2, 3, 4, 10, 11, 12, 13, 20, 21, 22, 30, 40.
  En comparant par exemple la proposition P = [2, 3, 3, 0] avec le code secret S = [3, 1, 3, 2] on obtient un score égal à 12 :

S =	3	1	3	2
P =	2	3	3	0

Score 12

- Les principes précédents peuvent être facilement étendus à des Mastermind comportant jusqu'à 10 couleurs et plus de 4 trous.
- Dans la suite du texte nous désignerons par X[i] l'élément de rang i de la liste X. Dans l'exemple précédent : S[0]=3 et P[3]=0.

2.1. Une première situation : Le codeur est l'ordinateur – le décodeur est humain

À l'instar de la construction des objets en Lego, nous allons élaborer les algorithmes en assemblant quelques « briques » décrites au préalable.

Brique Nº 1 : Choix du code secret

Le choix du code secret est aisé lorsqu'on dispose d'une fonction qui donne un entier aléatoire compris entre deux bornes a et b. Nous noterons cette fonction : *Alea*.

Fonction ChoixCode(nbTrous, nbCoul)
nbCoul désigne le nombre de couleurs autorisées *nbTrous désigne le nombre de trous.* *[] désigne une liste vide.*
Secret =[] *répéter nbTrous fois* *x=Alea(0,nbCoul-1)* *ajouter x en queue de la liste Secret* *résultat=Secret*

Remarque. Amiens-Python simplifie encore la tâche en proposant la fonction *listeRandint*(a, b, p) qui construit une p-liste d'entiers aléatoires tous compris entre a et b. Il suffira alors d'écrire : *Secret=listeRandint(0,nbCoul-1,nbTrous)*

Brique Nº 2 : Calcul du score

Il s'agit de calculer le score d'une proposition P (celle du décodeur) face au code secret S (détenu par le codeur).Voici une façon de procéder :

Le nombre r de fiches rouges peut être obtenu facilement en parcourant en parallèle les deux listes. Pour le nombre b de fiches blanches on pourra procéder comme suit : si la couleur i apparaît n_i fois dans S et m_i fois dans P alors elle est commune $\min(n_i, m_i)$ fois dans les deux codes. Il suffit alors de calculer la somme de ces minima pour l'ensemble des couleurs possibles. Puisque cette somme décompte également les couleurs bien placées, il convient de retrancher le nombre de fiches rouges. D'où :

$$b = \sum_{i=0}^{nbCoul-1} \min(n_i, m_i) - r.$$

Exemple de mise en œuvre pour :

S =	3	1	3	2
P =	2	3	3	0

Un balayage des deux listes nous donne une seule correspondance même place/même couleur pour la couleur 3.	$S=$ 3 1 **3** 2 $P=$ 2 3 **3** 0 *Nombre de fiches rouges : r=1*

On calcule ensuite les occurrences de chaque couleur dans *S* et *P*

	couleurs					
S	*0*	*1*	*2*	*3*	*4*	*5*
occurrences	0	1	1	2	0	0

	couleurs					
P	*0*	*1*	*2*	*3*	*4*	*5*
occurrences	1	0	1	2	0	0

	couleurs						
	0	*1*	*2*	*3*	*4*	*5*	
min	0	0	1	2	0	0	**3**

Le nombre de fiches blanches est égal à :
$b = 3 - r = 3 - 1 = 2$

Remarque. Il est intéressant de constater que dans cette comparaison les rôles de S et de P sont symétriques :

$$Score(S,P) = Score(P,S).$$

D'où notre deuxième brique :

Fonction Score(S, P)
S et P désignent les deux listes à comparer

```
r=0
répéter pour i de 1 à nbTrous
        si S[i]==P[i] alors r=r+1
b=-r
répéter pour i de 0 à nbCoul-1
        n=0
        m=0
        répéter pour j de 0 à nbTrous-1
                si S[j]=i alors n=n+1
                si P[j]=i alors m=m+1
        si n<m alors b=b+n
                sinon b=b+m
résultat=10*r+b
```

Là encore Python facilite les choses grâce aux fonctions sophistiquées de manipulation des listes.

En assemblant ces briques, nous obtenons un premier algorithme :

L'ordinateur est codeur
nbCoul=6 *nbTrous=4* *sc=0* *S=ChoixCode(nbCoul, nbTrous)* *répéter tant que sc<40* *P=demander la proposition du décodeur* *sc=Score(S, P)*

2.2. Seconde situation : l'ordinateur décode

Le principal atout de l'ordinateur est sans doute sa vitesse de calcul, il est donc possible d'envisager de passer en revue de manière exhaustive l'ensemble des codes possibles.
Voici l'idée centrale des algorithmes qui suivent : supposons que la proposition P a obtenu le score *sc* en la comparant au code secret S c'est-à-dire : *Score*(S,P) = *sc*. Si le décodeur fait alors l'hypothèse qu'une proposition P′ est la bonne (il fait donc l'hypothèse que P′ = S) on aura nécessairement *Score*(P′,P) = *sc*, ce qui lui permettra d'éliminer tous les candidats tels que *Score*(P′,P) ≠ *sc*.
D'où un premier algorithme :

Algorithme N°1(utilisé par le décodeur) ***(4 trous-6 couleurs)*** *La liste secrète est notée S.*
1. Puisqu'au départ toutes les listes sont équiprobables, en choisir une, P_0, au hasard. 2. Soit sc_0 son score : $sc_0=Score(S,P_0)$. *(*Ce score est calculé par le codeur*)*. 3. $n=1$ 4. Passer en revue les *listes candidates* à partir de [0, 0, 0, 0], … 5. Choisir comme nouvelle proposition P_n la première donnant les mêmes scores en la comparant avec les propositions précédentes. C'est-à-dire vérifiant : $\forall i \in [\![0, n-1]\!],\ Score(P_n, P_i) = sc_i$ 6. Soit $sc_n=Score(S,P_n)$. *(*Ce score est calculé par le codeur*)*. 7. Si $sc_n=40$, P_n est la solution cherchée, sinon, poursuivre le processus en augmentant n de 1 et en retournant à l'étape 5.

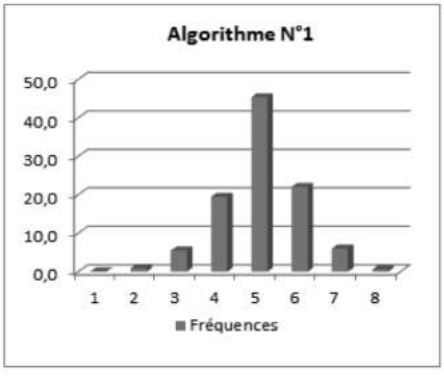
Ce procédé garantit le succès puisque le code cherché figure parmi les 1 296 possibilités éventuellement passées en revue.
L'ordinateur n'est pas malicieux, il en est bien incapable…
Pour tester cet algorithme et nous éviter la fastidieuse phase de l'évaluation des propositions par le codeur il n'y a donc aucun inconvénient à confier les deux tâches : codage et décodage à la même machine !

2.3. Une première preuve par ordinateur (par recherche exhaustive)

L'algorithme précédent garantit le succès quel que soit le code secret choisi mais en combien de coups ? Nous allons *démontrer* qu'il nécessite *au plus 8 propositions*. Pour cela il suffit de construire un programme qui passe en revue l'ensemble des codes secrets possibles et qui retient pour chaque code, le nombre de propositions nécessaires.
Le programme *Mastermind-3-Preuve* est constitué d'une boucle principale qui examine tous les 1 296 codes secrets de [0, 0, 0, 0] à [5, 5, 5, 5] en appliquant l'algorithme précédent. La proposition initiale arbitrairement retenue est toujours la même : [0, 0, 1, 1].

Voici les résultats obtenus :

Nombre de coups nécessaires	Effectifs	Fréquences en %	Fréquences cumulées
1	1	0,1	0,1
2	12	0,9	1
3	71	5,5	6,5
4	253	19,5	26
5	588	45,4	71,4
6	286	22,1	93,5
7	78	6,0	99,5
8	7	0,5	100
	1 296	100	



On constate que dans 93% des cas, au plus 6 essais suffisent.
On trouve qu'en moyenne 5 essais permettent de trouver le code secret.

Au vu de ce tableau nous pouvons énoncer un premier résultat :

Proposition 1
Au jeu de Mastermind (4 trous – 6 couleurs) il existe une stratégie gagnante permettant de découvrir le code secret en au plus 8 propositions.

3. L'algorithme de Knuth : le « five guess »

Nous nous proposons de démontrer que dans le cas du jeu classique 4 trous – 6 couleurs, le code secret peut être trouvé en 5 tentatives ou moins, d'où le nom *five–guess* (cinq conjectures) donné à cet algorithme.

3.1. Principe dans le cas classique

Si dans le programme précédent, on change de proposition initiale, les distributions ne sont plus les mêmes… Voici par exemple les résultats obtenus avec la proposition initiale : [1, 2, 3, 4]

Nombre de coups nécessaires	Effectifs	Fréquences en %	Fréquences cumulées
1	1	0,1	0,1
2	13	1	1,1
3	92	7,1	8,2
4	337	26	34,2
5	540	41,7	75,9
6	251	19,4	95,3
7	57	4,3	99,7
8	5	0,4	100
	1 296	100	

Ce tableau ne remet pas en cause le résultat précédent mais il suggère que la proposition initiale a une influence sur la suite. D'où une question légitime : Parmi toutes les propositions candidates, y-a-t-il des choix plus judicieux que d'autres ? Nous allons essayer de répondre à cette question.

Il semble assez naturel de retenir, parmi tous les candidats restants à une étape donnée, celui (ou l'un de ceux) qui minimise le nombre de candidats restants.
L'idée de D. Knuth consiste à choisir à chaque étape, *une proposition qui minimise le maximum (minimax) de propositions restantes en tenant compte des 14 scores possibles*.

Brique N°4 : Trouver une proposition optimale.

MeilleureProposition(Candidat, ListeComplète)

Cette fonction détermine une proposition optimale (suivant le principe minimax) à une étape donnée.
Candidat désigne la liste des candidats restants à une étape donnée.
ListeComplète désigne l'ensemble des 1 296 possibilités.

L'idée est d'attribuer à chaque proposition (parmi les 1296) un poids, puis de retenir une proposition de poids minimum. Voici comment procéder :

répéter pour chaque possibilité $P \in ListeComplete$ *:*
  répéter pour chaque score possible
  $s \in \{0,1,2,3,4,10,11,12,13,20,21,22,30,40\}$
    déterminer le nombre c de candidats obtenant le score s en les comparant avec P.
  attribuer à la proposition P un poids égal au plus grand des 14 nombres précédents.
choisir comme nouvelle proposition un minimax, c'est-à-dire l'une de celles (la première rencontrée par exemple) ayant le plus petit poids.

Algorithme N°2 : Five-Guess de KNUTH

construire la liste ListeComplète des 6^4 propositions possibles.
dupliquer cette liste dans la liste Candidats.
choisir une meilleure proposition initiale
P=MeilleureProposition(Candidats,ListeComplète)
calculer $sco = Score(S, P)$ *(rôle du codeur)*
répéter tant que $sco \neq 40$
    mettre à jour la liste Candidats en éliminant tous les éléments C vérifiant $Score(C, P) \neq sco$.
    choisir une meilleure proposition parmi les candidats restants :
    P=MeilleureProposition(Candidats,ListeComplète)
    calculer $sco = Score(S, P)$

Notons au passage que si à chaque étape on retient une proposition optimale, on n'a cependant aucune garantie quant à l'optimisation globale du processus.

3.2. Détails dans le cas 3 trous – 3 couleurs

Voici ce que donne la mise en œuvre de l'algorithme précédent dans le cas 3 trous-3 couleurs.
Le code secret choisi arbitrairement pour cet exemple est : [1,0,2]
Il y a 3^3 soit 27 propositions possibles.
Si la proposition était [0,0,0] : parmi les 27 candidats

- Il y en a 8 à savoir [1,1,1], [1,1,2], [1,2,1], [1,2,2], [2,1,1], [2,1,2], [2,2,1], [2,2,2] qui obtiennent le score 0.
- Il n'y en a aucun qui obtient le score 1
- Il n'y en a aucun qui obtient le score 2
- Il n'y en a aucun qui obtient le score 3
- Il y en a 12 à savoir [0,1,1], [0,1,2], [0,2,1], [0,2,2], [1,0,1], [1,0,2], [2,0,1], [2,0,2], [1,1,0], [1,2,0], [2,1,0], [2,2,0] qui obtiennent le score 10
- Il n'y en a aucun qui obtient le score 11
- Il n'y en a aucun qui obtient le score 12
- Il y en a 6 à savoir [0,0,1], [0,0,2], [0,1,0], [0,2,0], [1,0,0], [2,0,0], qui obtiennent le score 20.
- Il y en a une à savoir [0,0,0] qui obtient le score 40.

On obtient ainsi la première ligne du tableau ci-dessous. Le poids attribué à la proposition [0,0,0] est la plus grande des valeurs précédentes à savoir 12.
Ainsi : En choisissant [0,0,0] comme première proposition (parmi les 27) il ne restera qu'au plus 12 candidats en lice.

On procède de manière analogue pour toutes les 26 autres propositions.

		Scores possibles									
		00	01	02	03	10	11	12	20	30	*poids*
Liste Complète	[0,0,0]	8	0	0	0	12	0	0	6	1	12
	[0,0,1]	1	4	3	0	6	4	2	6	1	6
	[0,0,2]	1	4	3	0	6	4	2	6	1	6
	[0,1,0]	1	4	3	0	6	4	2	6	1	6
	[0,1,1]	1	4	3	0	6	4	2	6	1	6
	[0,1,2]	0	0	6	2	3	6	3	6	1	6
	[0,2,0]	1	4	3	0	6	4	2	6	1	6
	[0,2,1]	0	0	6	2	3	6	3	6	1	6
	[0,2,2]	1	4	3	0	6	4	2	6	1	6
	[1,0,0]	1	4	3	0	6	4	2	6	1	6
	[1,0,1]	1	4	3	0	6	4	2	6	1	6
	[1,0,2]	0	0	6	2	3	6	3	6	1	6
	[1,1,0]	1	4	3	0	6	4	2	6	1	6
	[1,1,1]	8	0	0	0	12	0	0	6	1	12
	[1,1,2]	1	4	3	0	6	4	2	6	1	6
	[1,2,0]	0	0	6	2	3	6	3	6	1	6
	[1,2,1]	1	4	3	0	6	4	2	6	1	6
	[1,2,2]	1	4	3	0	6	4	2	6	1	6
	[2,0,0]	1	4	3	0	6	4	2	6	1	6
	[2,0,1]	0	0	6	2	3	6	3	6	1	6
	[2,0,2]	1	4	3	0	6	4	2	6	1	6
	[2,1,0]	0	0	6	2	3	6	3	6	1	6
	[2,1,1]	1	4	3	0	6	4	2	6	1	6
	[2,1,2]	1	4	3	0	6	4	2	6	1	6
	[2,2,0]	1	4	3	0	6	4	2	6	1	6
	[2,2,1]	1	4	3	0	6	4	2	6	1	6
	[2,2,2]	8	0	0	0	12	0	0	6	1	12

Le poids minimal rencontré est 6.
Parmi les propositions ayant un poids 6 on en garde une (la première rencontrée fera l'affaire).
Nous proposerons donc [0,0,1] au codeur qui nous indiquera un score de 11 (puisque le code secret est [1,0,2]).
Un balayage exhaustif montre que parmi les 27 candidats, il n'y en a que 4 qui comparés à [0,0,1] donnent un score *sco* = 11.
Il s'agit de : [0,1,2], [0,2,0], [1, 0, 2], [2,0,0].

On recommence alors le processus avec ces 4 candidats…

		SCORES									
		00	01	02	03	10	11	12	20	30	*poids*
CODES	[0,0,0]	0	0	0	0	2	0	0	2	0	2
	[0,0,1]	0	0	0	0	0	4	0	0	0	4
	[0,0,2]	0	0	0	0	0	0	2	2	0	2
	[0,1,0]	0	0	1	0	0	1	0	2	0	2
	[0,1,1]	0	1	1	0	1	0	0	1	0	1
	[0,1,2]	0	0	1	0	0	1	1	0	1	1
	[0,2,0]	0	0	1	0	0	1	1	0	1	1
	[0,2,1]	0	0	1	1	0	0	1	1	0	1
	[0,2,2]	0	0	1	0	0	1	0	2	0	2
	[1,0,0]	0	0	1	0	0	1	0	2	0	2
	[1,0,1]	0	1	1	0	1	0	0	1	0	1
	[1,0,2]	0	0	1	0	0	1	1	0	1	1
	[1,1,0]	0	0	0	0	2	2	0	0	0	2
	[1,1,1]	2	0	0	0	2	0	0	0	0	2
	[1,1,2]	0	2	0	0	0	0	0	2	0	2
	[1,2,0]	0	0	0	1	0	1	1	1	0	1
	[1,2,1]	0	1	1	0	1	1	0	0	0	1
	[1,2,2]	0	1	0	0	1	1	0	1	0	1
	[2,0,0]	0	0	1	0	0	1	1	0	1	1
	[2,0,1]	0	0	1	1	0	0	1	1	0	1
	[2,0,2]	0	0	1	0	0	1	0	2	0	2
	[2,1,0]	0	0	0	1	0	1	1	1	0	1
	[2,1,1]	0	1	1	0	1	1	0	0	0	1
	[2,1,2]	0	1	0	0	1	1	0	1	0	1
	[2,2,0]	0	0	2	0	0	0	0	2	0	2
	[2,2,1]	0	0	2	0	2	0	0	0	0	2
	[2,2,2]	0	0	0	0	4	0	0	0	0	4

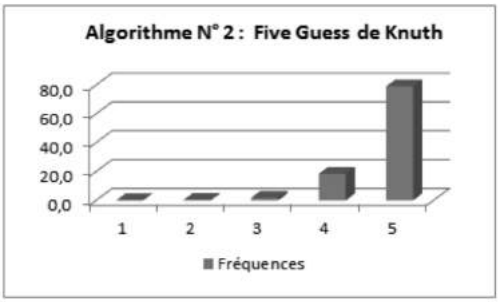
Le poids minimal rencontré est 1.
Parmi les propositions ayant un poids 1 on garde la première à savoir [0,1,1].
Nous proposerons donc [0,1,1] au codeur qui indiquera un score de 02.
Parmi les 4 candidats restants, il n'y en a qu'un qui comparé à [0,1,1] obtient le score *sco* = 02.
Il s'agit de la solution : [1, 0, 2]

Le problème a été résolu en deux étapes !

3.3. Preuve du five-guess.

Revenons au cas classique 4 trous – 6 couleurs. En passant en revue l'ensemble des codes possibles, donc en procédant de manière analogue à celle du §2.3 le programme Mastermind-4-Preuve nous permet de dresser un nouveau tableau statistique concernant le nombre de tentatives nécessaires pour trouver la solution.

Nombre de coups nécessaires	Effectifs	Fréquences en %	Fréquences cumulées
1	1	0,1	0,1
2	6	0,5	0,6
3	25	1,9	2,5
4	239	18,4	20,9
5	1 025	79,1	100
	1 296	100	



On constate que dans 21% des cas, au plus 4 essais suffisent.
On trouve qu'en moyenne 4,7 essais permettent de trouver le code secret.

Au vu de ce tableau nous obtenons un second résultat :

Proposition 2
Au jeu de Mastermind (4 trous – 6 couleurs) l'algorithme de Knuth permet de découvrir le code secret en au plus 5 propositions.

4. Épilogue

En 1993, Mami Koyama et Tony W. Lai ont trouvé un algorithme permettant de déterminer le code secret avec un nombre moyen de tentatives de 4,34 mais avec une situation exceptionnelle (une seule !) nécessitant 6 essais.

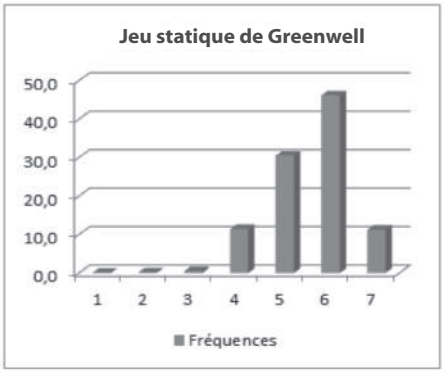
La mise en œuvre de l'algorithme de Knuth est inaccessible au joueur humain, sans recours à l'ordinateur. En 1995, J. Stuckman et Guo-Qiang Zhang ont démontré que le problème du Mastermind général était *NP-complet*. Autrement dit le temps d'exécution de l'algorithme précédent est exponentiel ce qui le rend inutilisable même pour des valeurs peu importantes de *nbCoul* et *nbTrous*.

Signalons enfin qu'il existe plusieurs séries de propositions qui rendent le jeu « statique » en ce sens qu'une telle série permet de déterminer de manière certaine (au prix d'un balayage systématique) la solution dans tous les cas.
Citons celle de D.L. Greenwell qui comporte 6 propositions :

[0,1,1,0], [1,2,4,3], [2,2,0,0], [3,4,1,3], [4,5,4,5], [5,5,3,2]

Voici les résultats correspondants :

Nombre de coups nécessaires	Effectifs	Fréquences en %	Fréquences cumulées
1	1	0,08	0,08
2	2	0,15	0,23
3	6	0,46	0,69
4	148	11,42	12,11
5	395	30,48	42,59
6	598	46,14	88,73
7	146	11,27	100
	1296	100	



On constate que dans 89% des cas, au plus 6 essais suffisent.
On trouve qu'en moyenne 5,6 essais permettent de trouver le code secret.

Références

D.E. Knuth, *The computer as Master Mind*, Journal of Recreational Mathematics, Vol 9(1), 1976-77,
consultable sur http://colorcode.laebisch.com/links/Donald.E.Knuth.pdf.

ANNEXE 1

Liste des programmes Python disponibles sur le site de l'APMEP

Nom	Description
Mastermind-1.py	L'ordinateur est codeur. A chaque proposition saisie au clavier (sous la forme numérique) le programme répond en indiquant le score obtenu par la proposition. Aucun contrôle de saisie n'est effectué.
Mastermind-2.py	L'ordinateur est décodeur. Mise en œuvre du premier algorithme. Un code secret est choisi aléatoirement. Le programme décode automatiquement.
Mastermind-2-1.py	Variante du programme précédent. L'ordinateur décode en utilisant le premier algorithme. Le codeur est l'utilisateur humain.
Mastermind-3-preuve.py	Le premier algorithme est testé pour les 1296 propositions du jeu 4-6. Ceci constitue la preuve du théorème. Le programme s'achève en affichant des statistiques et un diagramme en bâtons
Mastermind-4.py	L'algorithme de Knuth L'ordinateur décode le code secret choisi aléatoirement. Le programme permet d'exécuter un certain nombre de parties consécutives.
Mastermind-4-Preuve.py	La preuve du five-guess. Toutes les possibilités sont passées en revue
Mastermind-5.py	Ce programme étudie la série de « meilleures » propositions statiques : [0,1,1,0], [1,2,4,3], [2,2,0,0], [3,4,1,3], [4,5,4,5], [5,5,3,2] Cette suite est proposée dans cet ordre au décodeur pour chaque code secret possible. On procède de la même manière que dans l'algorithme de Knuth.

ANNEXE 2

Conversion d'un entier en base b

Fonction Base(n, b)
convertit l'entier n (écrit dans le système décimal) en base b . *quotient (a,b) désigne le quotient entier de la division de a par b.* *reste(a,b) désigne le reste de la division de a par b.*
q=quotient(n,b) *x=reste(n,b)* *p=1* *répéter tant que* $q \neq 0$ *p=10*p* *x=x+10*reste(q,b)* *q=quotient(q,b)* *résultat x*

Écriture d'un entier sous forme de liste

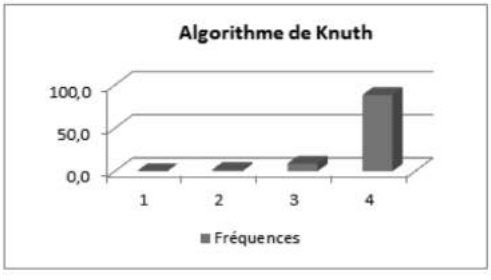
Proposition(n)
nbTrous désigne le nombre de trous. *adjq(L,d) désigne l'adjonction de l'élément de d en queue de liste L.* *Par exemple si L=[1, 2] après adjq(L,3), L aura pour valeur [1, 2, 3]*
L=[] *répéter pour i de 1 à nbTrous* *p= puissance(10,nbTrous- i)* *d=quotient(n,p)* *L=adjq(L,d)* *n=n-d*p* *résultat L*

ANNEXE 3

L'algorithme de Knuth dans diverses situations.

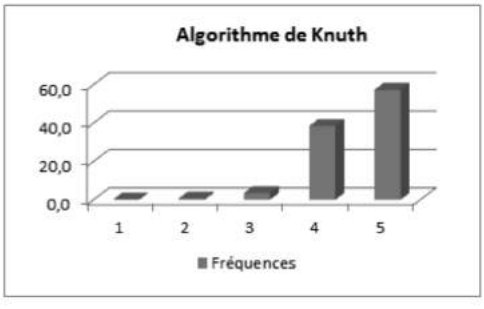
4 trous - 4 couleurs

Nombre de coups nécessaires	Effectifs	Fréquences en %	Fréquences cumulées
1	1	0,4	0,4
2	5	1,9	2,3
3	23	9	11,3
4	227	88,7	100
	256	100	



4 trous - 5 couleurs

Nombre de coups nécessaires	Effectifs	Fréquences en %	Fréquences cumulées
1	1	0,2	0,2
2	4	0,6	0,8
3	22	3,5	4,3
4	240	38,4	42,7
5	358	57,3	100
	625	100	



4 trous - 7 couleurs

Nombre de coups nécessaires	Effectifs	Fréquences en %	Fréquences cumulées
1	1	0,04	0,1
2	2	0,08	0,12
3	18	0,75	0,87
4	256	10,66	11.53
5	1 459	60,77	72.3
6	665	27,7	100
	2 401	100