

MÉTROPOLE – 2026 – sujet1

EXERCICE 2 (6 POINTS)

Cet exercice porte sur le binaire, les graphes, la récursivité.

Le jeu du Recto-Verso, de la famille des solitaires, se compose de neuf jetons disposés sur un plateau carré. Le plateau a 3 lignes et 3 colonnes. Chaque jeton possède une face Recto, dessinée en couleur noire sur les figures, et une face Verso, dessinée en blanc.

PARTIE A : REPRÉSENTATION BINAIRE DES CONFIGURATIONS

On appelle *configuration* l'état du plateau. La figure 1 illustre un exemple de configuration.

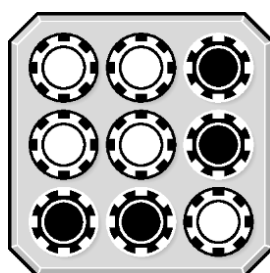


Figure 1. Illustration d'une configuration

1. Justifier que le nombre total de configurations possibles est 2^9 .

Le but du jeu est, depuis une configuration quelconque, de retourner tous les jetons sur leur face Recto (noire). Pour ce faire, le seul type d'opération autorisé est de choisir une ligne ou une colonne ou une diagonale, et d'en retourner tous les jetons. On appelle une telle opération un *coup*. La figure 2 illustre les huit différents coups possibles à partir d'une configuration donnée.

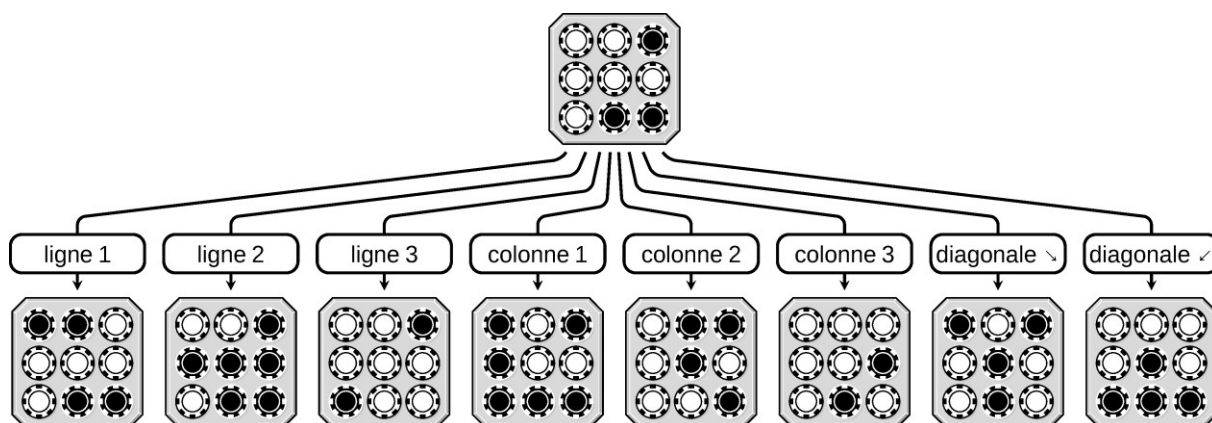


Figure 2. Les coups possibles depuis la configuration 67

2. Indiquer en justifiant si la séquence de coups suivante, jouée depuis la configuration donnée en figure 1, permet de mener à la victoire : ligne 1, ligne 2, colonne 2.

Une configuration est représentée, ligne par ligne et de haut en bas, par une liste de bits en considérant les jetons Recto comme des 1 et les jetons Verso comme des 0. Ainsi on lit la configuration initiale de la figure 2 comme la liste $[0, 0, 1, 0, 0, 0, 0, 1, 1]$.

On peut aussi lire la liste de bits d'une configuration comme l'écriture binaire d'un nombre entier. Pour l'exemple de la figure 2, la configuration initiale est donc aussi représentée par l'entier 67.

3. En détaillant le calcul, donner l'entier représentant la configuration illustrée en figure 1.
4. Implémenter une fonction représentant prenant en paramètre une liste de 0 et de 1 et renvoyant l'entier correspondant.

Dans la suite on considère à disposition une fonction binaire prenant réciproquement en paramètre un entier et renvoyant son écriture en binaire sous forme d'une liste de neuf 0 ou 1.

L'opérateur ou exclusif, noté $\oplus$, a pour table de vérité :

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

La fonction xor définie ci-dessous prend en paramètres deux bits b1, b2 (de type int) et renvoie le résultat de $b1 \oplus b2$.

```

1 def xor(b1, b2):
2     if b1 == b2:
3         return 0
4     else:
5         return 1

```

L'opérateur $\oplus$ peut être appliqué sur deux listes de même longueur l_x et l_y contenant plusieurs bits : pour chaque indice de ces listes, on applique le $\oplus$ aux deux bits positionnés à cet indice. Par exemple, si $l_x = [1, 1]$ et $l_y = [0, 1]$, alors $l_x \oplus l_y$ vaut $[1, 0]$, puisque $1 \oplus 0$ vaut 1, et $1 \oplus 1$ vaut 0.

5. Implémenter une fonction xor_etendu qui prend en paramètre deux listes de bits de même longueur l_x et l_y et renvoie la liste $l_x \oplus l_y$.

On peut observer que pour un bit X donné, l'opération $X \oplus 1$ a pour résultat l'inverse de X et l'opération $X \oplus 0$ a pour résultat X.

Si X représente un jeton, alors $X \oplus 1$ le retourne et $X \oplus 0$ le laisse inchangé.

Appliquer un coup à une configuration revient à lister les jetons retournés et les jetons non retournés. On peut donc représenter chaque coup par une liste de bits. Par exemple le coup « ligne 1 » est représenté par la liste $[1, 1, 1, 0, 0, 0, 0, 0, 0]$ et l'appliquer à la configuration 67 revient à calculer $[0, 0, 1, 0, 0, 0, 0, 1, 1] \oplus [1, 1, 1, 0, 0, 0, 0, 0, 0]$.

La figure 3 illustre les 8 coups possibles de la même manière au moyen du $\oplus$.

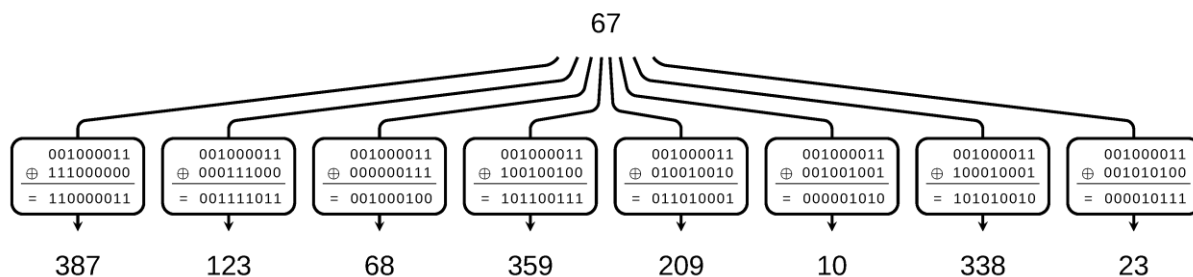


Figure 3. Les coups possibles depuis la configuration 67 appliqués par un xor. Par souci de lisibilité, la notation en liste est simplifiée.

On suppose à disposition le dictionnaire `coups_possibles` qui énumère chaque coup représenté de la sorte :

```
1 coups_possibles = {
2     "ligne 1"      : [1, 1, 1, 0, 0, 0, 0, 0, 0],
3     "ligne 2"      : [0, 0, 0, 1, 1, 1, 0, 0, 0],
4     "ligne 3"      : [0, 0, 0, 0, 0, 0, 1, 1, 1],
5     "colonne 1"    : [1, 0, 0, 1, 0, 0, 1, 0, 0],
6     "colonne 2"    : [0, 1, 0, 0, 1, 0, 0, 1, 0],
7     "colonne 3"    : [0, 0, 1, 0, 0, 1, 0, 0, 1],
8     "diagonale 1"  : [1, 0, 0, 0, 1, 0, 0, 0, 1],
9     "diagonale 2"  : [0, 0, 1, 0, 1, 0, 1, 0, 0]
10 }
```

On souhaite implémenter une fonction qui prend en paramètre un entier `config` représentant une configuration et qui renvoie la liste des entiers correspondant à toutes les configurations accessibles en un seul coup.

```
1 def configurations_suivantes(config):
2     config_bin = ...
3     voisins = []
4     for ... in ...:
5         ... # ligne facultative
6         ... = xor_etendu(..., ...)
7         voisins.append(representant(...))
8     return voisins
```

6. Recopier et compléter la fonction `configurations_suivantes`.

PARTIE B : GRAPHE DES CONFIGURATIONS

On constate qu'il semble impossible d'atteindre la victoire depuis la configuration représentée par l'entier 4. On appelle *configuration perdante* toute configuration pour laquelle il n'existe aucune suite de coups menant à la victoire (c'est-à-dire à tous les jetons côté Recto). Inversement, on appelle *configuration gagnante* toute configuration qui n'est pas perdante.

La configuration représentée par l'entier 4 est effectivement perdante. Pour s'en convaincre on construit le graphe des configurations du jeu Recto-Verso.

Les sommets de ce graphe sont les différentes configurations possibles et il existe une arête $s_1 \rightarrow s_2$ d'un sommet s_1 vers un sommet s_2 si et seulement s'il est possible de passer de la configuration s_1 à la configuration s_2 en jouant un coup autorisé.

7. Montrer que s'il existe une arête $x \rightarrow y$ dans le graphe, alors il existe également une arête $y \rightarrow x$.

En Python, on choisit de représenter ce graphe par un dictionnaire construit grâce à la fonction `configurations_suivantes` :

```
1 graphe = {  
2     0 : [...],  
3     ...  
4     67 : [387, 123, 68, 359, 209, 10, 338, 23],  
5     ...  
6 }
```

8. Une représentation par matrice d'adjacence aurait aussi pu être envisagée. Justifier le choix du dictionnaire en comparant pour les deux structures le nombre d'entiers nécessaires pour stocker en mémoire le graphe des configurations. On pourra laisser ce nombre écrit sous la forme d'une puissance de 2.

La fonction `parcours_profondeur` ci-dessous implémente un parcours en profondeur d'abord. Elle prend en paramètres la représentation d'un graphe, un entier `configuration` qui indique le sommet que l'on est en train de visiter, et une liste `vus` de sommets déjà visités par le parcours. Cette fonction effectue le parcours en rajoutant les sommets visités à la liste `vus`.

```
1 def parcours_profondeur(graphe, configuration, vus):  
2     vus.append(...)  
3     for s in ...:  
4         if s not in vus:  
5             parcours_profondeur(..., ..., ...)
```

9. Recopier et compléter la fonction `parcours_profondeur`.
10. En utilisant la fonction `parcours_profondeur`, écrire en Python une suite d'instructions permettant de vérifier que la configuration représentée par l'entier 4 est bien perdante.
11. Indiquer et justifier quel parcours de graphe (en largeur ou en profondeur) est le plus approprié pour trouver un chemin partant d'une configuration gagnante et menant à la victoire en un minimum de coups.