ACADÉMIE DE TOULOUSE
Liberté
Égalité
Fraternité

ACADÉMIE DE VERSAILLES
Liberté
Égalité
Fraternité

ACADÉMIE DE MONTPELLIER
Liberté
Égalité
Fraternité

Olympiades numérique et sciences informatiques (NSI)

Académies de Toulouse, Versailles et Montpellier - 9 avril 2025

Cette épreuve est individuelle et dure trois heures.

Aucun document n'est autorisé. Aucun matériel électronique n'est autorisé, en particulier **les calculatrices et les ordinateurs ne sont pas autorisés**.

Le seul langage de programmation autorisé dans cette épreuve est Python. Le code proposé doit impérativement être proprement indenté afin d'éviter toute ambiguïté quant à sa validité. De plus, pour les mêmes raisons, on veillera à ne pas écrire le code d'une fonction à cheval sur deux pages. Plus généralement, on veillera à la lisibilité, la simplicité et l'efficacité du code proposé. De plus, l'utilisation d'identifiants significatifs pour les fonctions et les variables, ainsi que l'emploi judicieux de commentaires dans le code seront appréciés.

Ce sujet comporte deux exercices totalement indépendants. Ces deux exercices doivent être **traités sur des copies séparées**.

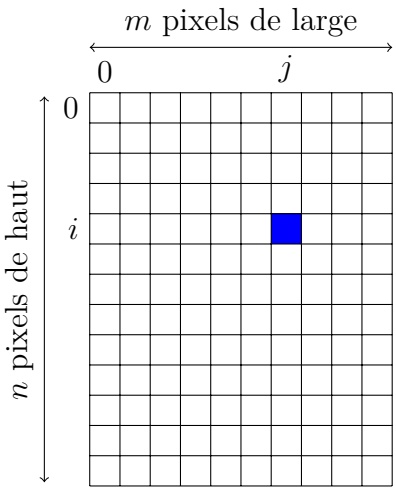
À tout moment vous pouvez faire appel à une fonction définie plus haut dans l'exercice, même si vous n'avez pas traité la question correspondante. Si vous ne pouvez pas formuler une réponse complète à une question, il vous est néanmoins conseillé d'exposer le bilan de vos pistes de recherche.

Le sujet comporte 15 pages. Avant de commencer, vérifiez que votre exemplaire est complet.

Exercice 1 : Compression d'images

Il existe plusieurs façons d'encoder les images dans des fichiers. Dans cet exercice nous verrons plusieurs de ces codages, et nous nous intéresserons à la compression d'image, c'est-à-dire à la transformation d'un fichier codant une image en un fichier codant la même image mais de manière à utiliser moins d'octets en mémoire.

On suppose dans tout l'exercice que les images encodées sont rectangulaires. Une manière classique d'encoder une telle image est de la représenter comme une grille de points colorés appelés **pixels**. Une image de largeur de m pixels et de hauteur de n pixels peut être vue comme un tableau de $n \times m$ pixels ayant n lignes et m colonnes. Les lignes sont numérotées de haut en bas, de 0 à $n-1$, et les colonnes de gauche à droite, de 0 à $m-1$. Ainsi le pixel du coin en haut à gauche de l'image correspond à la case d'indice $(0,0)$ de ce tableau. Le pixel à la position (i,j) correspond à la case d'indice (i,j) du tableau, c'est-à-dire celle sur la i-ième ligne et j-ième colonne. En Python, on peut coder une ligne de ce tableau par une liste de pixels, et l'intégralité du tableau comme la liste de ses lignes. Ainsi une image est une liste de listes de pixels.



Chaque pixel de l'image a une **couleur** qui est représentée en Python différemment selon les cas.

- Lorsque l'image est en **noir et blanc**[1], la couleur peut être `0`, si le pixel est noir, ou `1`, si le pixel est blanc.
- Lorsque l'image est en 8 niveaux de gris, la couleur est un entier compris entre 0 et 7, 0 correspond à un pixel noir, 1 à un pixel gris très foncé... 6 à un pixel gris très clair, et 7 à un pixel blanc.
- Lorsque l'image est en couleurs, la couleur d'un pixel est en général un triplet d'entiers compris entre 0 et 255 : le premier donne le niveau de rouge, le deuxième le niveau de vert et le troisième celui de bleu. On parle alors de triplet RGB (pour *Red Green Blue* en anglais). Par exemple, un pixel de couleur $(0,0,255)$ est bleu, un pixel de couleur $(0,255,0)$ est vert, un pixel de couleur $(0,255,255)$ est cyan et un pixel de couleur $(250,150,0)$ est orange (une certaine nuance de orange). De plus, quand les trois niveaux sont à 0, le pixel est noir (absolument pas lumineux) et lorsque les trois niveaux sont à 255, il est blanc.

En guise d'exemple, on définit ci-dessous trois images en Python, qui sont représentées sur la figure 1 : `imgA` en noir et blanc, `imgB` en niveau de gris et `imgC` en couleurs.

```
1 imgA = [ [0,1,1,1,1], [1,0,1,1,1], [1,1,0,1,1], [1,1,1,0,1], [1,1,1,1,0] ]
2 imgB = [ [0,1,2,3,4,5,6,7], [0,1,2,3,4,5,6,7], [0,1,2,3,4,5,6,7],
  ↪  [0,1,2,3,4,5,6,7] ]
3 imgC = [ [(255,0,0), (255,0,0), (255,0,0)], [(255,255,255), (255,255,255),
  ↪  (255,255,255)]]
```

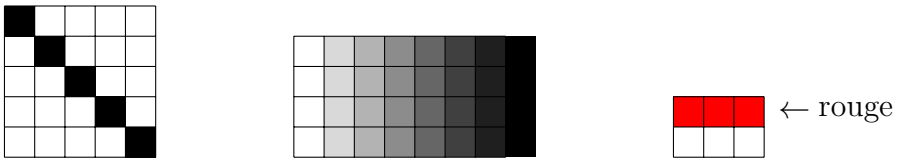


FIGURE 1 – Représentation des images `imgA`, `imgB` et `imgC` (de gauche à droite)

1. Les images dites "en noir et blanc" dans le langage courant sont en fait des images codées en niveau de gris

Question 1
Indiquer ce que valent alors `len(imgB)` et `len(imgB[0])`.

Afin de créer des images en Python, on utilise une fonction `creer_image` qui crée une image unie de couleur et de dimensions données. La description complète de cette fonction peut être obtenue en tapant dans un terminal la commande `help(creer_image)` : celle-ci affiche la documentation suivante.

```
creer_image(n, m, c):
    """
    Parameters
    ----------
    n : int
        nb de lignes
    m : int
        nb de colonnes
    c : couleur

    Returns
    -------
    liste de listes correspondant à l'image à n lignes
    et m colonnes et remplie de pixels tous de couleur c
```

Exemple :
L'appel `creer_image(2, 1, (0, 0, 200))` renvoie la liste `[[(0, 0, 200)], [(0, 0, 200)]]` qui correspond à une image à 2 lignes et 1 colonne contenant deux pixels, tous les deux de la même nuance de bleu.

Question 2
Recopier et compléter (sur la copie) le code de la fonction `creer_image` donné ci-dessous.

```
def creer_image(n, m, c):
    img = []
    for i in range(....):
        nouvelle_ligne = []
        for j in range(....):
            nouvelle_ligne.append(c)
        img.append(....)
    return img
```

Question 3

a) Écrire l'instruction qui permet de créer `imgD` une image en couleurs entièrement blanche de largeur 30 pixels et de hauteur 10 pixels.

b) On souhaite mettre un pixel rouge dans le coin en haut à droite de l'image `imgD`. L'exécution de l'instruction `imgD[0][30] = (255,0,0)` déclenche une erreur. Corriger l'instruction.

c) Écrire les instructions qui transforment [2] l'image `imgD` en un drapeau français.

2. On rappelle qu'on ne peut pas modifier les composantes d'un tuple, ainsi on ne peut pas modifier le niveau de rouge, de vert ou de bleu d'un pixel. On peut en revanche affecter un nouveau triplet RGB à ce pixel.

Partie 1 : Les couleurs d'une image

Le nombre de couleurs possibles avec des triplets RGB est 256^3 car chaque niveau (rouge, vert ou bleu) a 256 valeurs possibles. En vue de la compression, on souhaite réduire ce nombre de couleurs possibles afin d'augmenter les chances de trouver de grandes zones d'une même couleur. On propose ici deux transformations : le passage en noir et blanc (qui réduit le nombre de couleurs possibles à 2), puis le passage en 8 couleurs (qui réduit le nombre de couleurs possibles à 8 comme son nom l'indique).

Passage en noir et blanc. On souhaite d'abord transformer une image en couleurs en une image en noir et blanc. Pour cela, on effectue sur chaque pixel la procédure suivante.

- On calcule la moyenne des niveaux de rouge, de vert et de bleu. Cette moyenne est donc un réel appartenant à $[0; 255]$.
- Si celle-ci est strictement inférieure à 128, le pixel prend la valeur 0, sinon il prend la valeur 1.

Question 4

Écrire le code d'une fonction `noir_ou_blanc` qui a pour paramètre un réel `x` et qui renvoie 0 si `x` est strictement inférieur à 128 et 1 sinon.

Question 5

Écrire le code d'une fonction `passer_en_nb` qui a pour paramètre une image `img` en couleurs et qui renvoie l'image en noir et blanc correspondante. Cette fonction ne doit pas modifier l'image en entrée, elle doit créer une nouvelle image.

Passage en 8 couleurs. Les 8 couleurs que l'on s'autorise ici sont les $8 = 2^3$ triplets de 0 et 1. À nouveau les trois composantes de tels triplets représentent les niveaux respectifs de rouge, de vert et de bleu, mais chaque niveau ne peut être que 0 ou 1. On calcule le triplet de 0 et 1 correspondant à un triplet RGB niveau par niveau : si le niveau a une valeur strictement inférieure à 128, il prend la valeur 0, sinon la valeur 1.

Exemple :

Un pixel de couleur `(255, 100, 129)` prendra la valeur `(1, 0, 1)` lorsqu'on passe en 8 couleurs.

Question 6

Écrire le code d'une fonction `passer_en_8_couleurs` qui a pour paramètre une image en couleurs `img` (une liste de liste de triplets RGB) et qui renvoie l'image en 8 couleurs correspondante. Cette fonction ne doit pas modifier l'image en entrée, elle doit créer une nouvelle image.

Partie 2 : Compression en ligne d'une image en noir et blanc

Dans cette partie, on travaillera uniquement avec des images en noir et blanc, chaque pixel sera donc codé sur un bit et vaut soit `0` soit `1`. L'objectif de cette partie est d'écrire des fonctions permettant de compresser une telle image, puis de la décompresser pour revenir à l'image initiale.

Conversion image/ligne. Dans un premier temps, on souhaite stocker différemment l'image. L'objectif est de la mémoriser sous la forme d'une seule "grande" ligne contenant tous les pixels les uns à la suite des autres. On mémorise également le nombre de colonnes de l'image (afin de pouvoir reformer l'image de départ lors de la décompression).

Pour cela, on souhaite écrire une fonction `image_vers_ligne` qui a pour paramètre `img` une liste de listes de `0` ou `1` et qui renvoie un tuple dont la première valeur est le nombre de colonnes de l'image et la seconde valeur est une liste contenant tous les pixels.
Exemple :
L'appel `image_vers_ligne([[0,1,0],[1,1,1]])` renvoie le tuple `(3,[0,1,0,1,1,1])`.
L'appel `image_vers_ligne([[0,0,0,0],[1,1,1,1]])` renvoie le tuple `(4,[0,0,0,0,1,1,1,1])`.

Question 7
Écrire le code d'une telle fonction `image_vers_ligne`.

Pour pouvoir revenir à l'image initialement stockée sous la forme d'une liste de listes, on souhaite également écrire une fonction `ligne_vers_image` qui a pour paramètres un entier `nbc` correspondant au nombre de colonnes de l'image de départ et une liste `ligne01` composée de `0` et `1`, et qui renvoie l'image correspondante sous la forme d'une liste de listes de `0` et `1`.
Exemple :
L'appel `ligne_vers_image(3,[0,1,0,1,1,1])` renvoie la liste de listes `[[0,1,0],[1,1,1]]`.

Question 8
Écrire le code d'une telle fonction `ligne_vers_image`.

Compression et décompression de ligne. Afin de compresser une liste de `0` et de `1`, les éléments consécutifs de même valeur sont comptés et leur nombre est mémorisé dans une liste. Par exemple une liste contenant seulement huit `0` est représentée par la liste `[8]`, une liste contenant six `0` puis deux `1` est représentée par `[6,2]` et une liste alternant `0` et `1` quatre fois en commençant par `0` est représentée par `[1,1,1,1,1,1,1,1]`.

Pour réaliser cette compression on souhaite écrire une fonction `compte_0_1` qui a pour paramètre une liste `ligne01` composée de `0` et de `1` et qui renvoie une liste d'entiers constituée du nombre de `0` successifs dans `ligne01`, puis du nombre de `1` successifs suivant ces `0`, puis du nombre de `0` successifs suivant ces `1`, *etc.* On précise qu'**on commencera toujours par compter le nombre de `0`**, même si la liste commence par un `1`.
Exemple :
L'appel `compte_0_1([0,1,0,0,0,1,1])` renvoie la liste `[1,1,3,2]`.
L'appel `compte_0_1([1,1,1,0,1,1])` renvoie la liste `[0,3,1,2]`.

Question 9
Écrire le code de la fonction `compte_0_1`.

Afin de pouvoir revenir à la liste de `0` et de `1` initiale, on souhaite écrire une fonction `decompresse_ligne` qui a pour paramètre une liste d'entiers positifs `liste_entiers` et qui renvoie la liste contenant autant de `0` et de `1` que l'indiquent les valeurs de `liste_entiers`. On précise que la première valeur de `liste_entiers` correspond au nombre de `0` au début de la liste à renvoyer et peut donc être nul.
Exemple :
L'appel `decompresse_ligne([4,2,3,1])` renvoie `[0,0,0,0,1,1,0,0,0,1]`.
L'appel `decompresse_ligne([0,2,2])` renvoie `[1,1,0,0]`.

Question 10
Écrire le code d'une telle fonction `decompresse_ligne`.

Compression et décompression d'image. Grâce aux fonctions précédentes, on peut maintenant compresser une image en la transformant d'abord en ligne puis en compressant cette ligne. Afin de pouvoir décompresser, c'est-à-dire de pouvoir reformer l'image initiale, on veillera à stocker le nombre de colonnes de l'image avec cette ligne compréssée.

On souhaite donc écrire deux fonctions :

- une fonction `compresse_image` qui a pour paramètre `img` une image sous la forme d'une liste de listes de pixels (`0` ou `1`) et qui renvoie l'image compressée sous la forme d'un tuple à deux éléments, contenant d'une part le nombre de colonnes de l'image initiale et d'autre pat une liste d'entiers comptant le nombre de `0` et de `1` consécutifs ;
- une fonction `decompresse_image` qui a pour paramètres un entier `nbc` et une liste d'entiers `liste_entiers` correspondant au nombre de `0` et de `1` consécutifs et qui renvoie l'image sous la forme d'une liste de listes de pixels valant `0` ou `1`.

Exemple :
L'appel `compresse_image([[0,1,0],[1,1,1]])` renvoie `(3,[1,1,1,3])`.
L'appel `decompresse_image(3,[1,1,1,3])` renvoie `[[0,1,0],[1,1,1]]`.

Question 11
Écrire le code d'une telle fonction `compresse_image`.

Question 12
Écrire le code d'une telle fonction `decompresse_image`.

Partie 3 : Compression d'une image avec des rectangles couvrants

Dans cette partie, on cherche à compresser une image en découpant l'image en rectangles de même couleur d'aire maximale. Bien que cette technique de compression s'applique à tout type d'image, on l'illustre sur l'image en 8 couleurs `img1` définie ci-contre. La figure 2 illustre comment cette image sera découpée en rectangles.

```
img1 =  [
    [7, 7, 7, 1],
    [7, 7, 7, 1],
    [2, 2, 1, 1]
]
```

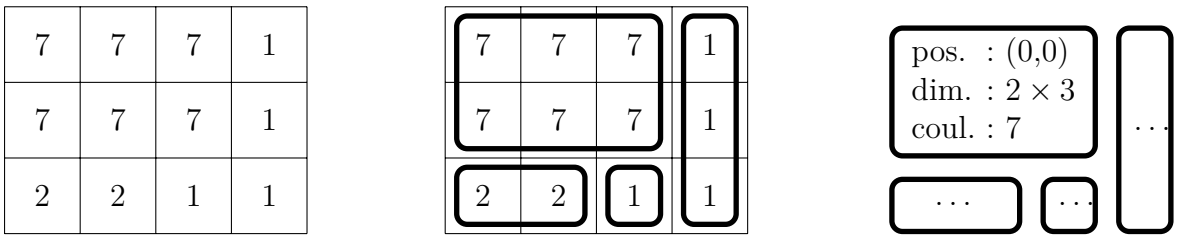


FIGURE 2

Plus grand rectangle uni. Afin d'obtenir un tel découpage de l'image, il faut pouvoir déterminer, à partir d'une position `pos` sur l'image, un rectangle dont le coin supérieur gauche est situé en `pos`, dont tous les pixels sont de la même couleur et d'aire la plus grande possible. On dira d'un tel rectangle qu'il est un plus grand rectangle uni (de couleur unie) en `pos`. Par exemple, sur l'image `img1`, le plus grand rectangle uni en `(0,0)` est le rectangle de largeur 3 pixels et de hauteur 2 pixels (il est donc d'aire $2 \times 3 = 6$). En effet ce rectangle est bien uni (de couleur `7`) et on ne peut pas trouver un rectangle uni en `pos` d'aire strictement supérieure à 6.

Question 13
Indiquer un plus grand rectangle uni en `(0,1)` pour l'image ci-contre. Dire si ce plus grand rectangle est unique et préciser ses dimensions (hauteur et largeur) en nombre de pixels ainsi que sa couleur.

6	3	3	3	4
6	3	3	6	4
6	3	4	6	4

Afin d'obtenir des plus grands rectangles unis, on souhaite d'abord écrire `dim_largeur_max`, une fonction qui a pour paramètres `img` une liste de listes de pixels, `pos` un tuple de deux entiers correspondant à une position valide dans `img` et `j_max` entier correspondant à un indice de colonne valide dans `img` et qui renvoie le nombre de pixels dans `img` de même couleur que celui à la position `pos` sur la même ligne et vers la droite jusqu'à la colonne `j_max` incluse. Pouvoir régler le paramètre `j_max` sera utile par la suite, mais dans un premier temps on peut imaginer qu' on appelle cette fonction avec l'indice de la dernière colonne de l'image en troisième argument, comme ci-dessous.
Exemple :
L'appel `dim_largeur_max(img1,(0,0),3)` renvoie `3`.
L'appel `dim_largeur_max(img1,(2,0),3)` renvoie `2`.
L'appel `dim_largeur_max(img1,(2,1),3)` renvoie `1`.

Question 14
Écrire le code d'une telle fonction `dim_largeur_max`.

On souhaite maintenant écrire une fonction `rectangle_max` qui a pour paramètres `img` une liste de listes de pixels, `pos` un tuple de deux entiers correspondant à une position valide dans `img` et qui renvoie un tuple d'entiers qui donne les dimensions (hauteur et largeur en nombre de pixels) d'un plus grand rectangle uni en `pos` dans `img`.

Question 15
Afin de tester le code qu'on proposera pour cette fonction, on prépare le jeu de tests ci-contre. Dire ce que permet de vérifier chacun des tests.

```
assert rectangle_max(img1,(0,0)) == (2,3)
assert rectangle_max(img1,(0,2)) == (2,1)
assert rectangle_max(img1,(0,3)) == (3,1)
assert rectangle_max(img1,(2,3)) == (1,1)
```

Question 16
Écrire le code d'une fonction `rectangle_max` correspondant à la description donnée plus haut.

Représentation des rectangles unis. Chaque rectangle uni peut être représenté en Python par un dictionnaire contenant 4 associations dont les clefs et les valeurs sont les suivantes :

- la valeur associée à la clef `"pos"` est un tuple de deux entiers indiquant la position de son coin supérieur gauche ;
- la valeur associée à la clef `"coul"` est la couleur de ses pixels ;
- la valeur associée à la clef `"ha"` est un entier indiquant sa hauteur en nombre de pixels ;
- la valeur associée à la clef `"la"` est un entier indiquant sa largeur en nombre de pixels.

Exemple :
Le premier rectangle de `img1` est représenté par `{"pos":(0,0), "coul":7, "ha":2, "la":3}`.

Compression et décompression. On souhaite maintenant procéder à la compression d'une image en la découpant en rectangles unis. Afin de pouvoir reformer l'image initiale lors de la décompression, on stocke les dimensions de l'image initiale en plus de la liste des rectangles du découpage. Ainsi une **image compressée** est représentée par un tuple composé :

- d'un entier indiquant la hauteur de l'image en nombre de pixels,
- d'un entier indiquant la largeur de l'image en nombre de pixels,
- d'une liste de dictionnaires représentant des rectangles unis.

On dit qu'un tel tuple `(ha,la,l_rect)` est une image compressée **valide** si les trois conditions suivantes sont vérifiées :

- les rectangles de `l_rect` sont tous dans l'image de dimensions `ha` × `la` ;
- les rectangles de `l_rect` ne se superposent pas ;
- les rectangles de `l_rect` couvrent tous les pixels de l'image de dimensions `ha` × `la` .

On souhaite écrire deux fonctions :

- une fonction `compression_rectangle` qui a pour paramètre une image `img` (sous la forme d'une de liste de listes de pixels), qui parcourt cette image ligne par ligne et en commençant par le coin supérieur gauche pour la découper en rectangles, et qui renvoie l'image compressée associée (sous la forme d'un tuple tel que décrit plus haut).
- une fonction `decompression_rectangle` qui a pour paramètre une image compressée valide et qui renvoie l'image décompressée correspondante.

Exemple :
L'appel `compression_rectangle(img1)` renvoie le tuple `ic1` suivant.

```
(3, 4,[ {"pos":(0,0), "coul":7, "ha":2, "la":3},
        {"pos":(0,3), "coul":1, "ha":3, "la":1},
        {"pos":(2,0), "coul":2, "ha":1, "la":2},
        {"pos":(2,2), "coul":1, "ha":1, "la":1}])
```

L'appel `decompression_rectangle(ic1)` renvoie alors l'image `img1` .

On remarque que le tuple `ic2` ci-dessous était aussi valide pour représenter `img1` (c'est-à-dire que `decompression_rectangle(ic2)` renvoie aussi `img1`), mais il ne correspond pas au découpage attendu vu l'ordre de parcours imposé dans la fonction `compression_rectangle` .

```
(3, 4,[ {"pos":(0,0), "coul":7, "ha":2, "la":3},
        {"pos":(0,3), "coul":1, "ha":2, "la":1},
        {"pos":(2,0), "coul":2, "ha":1, "la":2},
        {"pos":(2,2), "coul":1, "ha":1, "la":2}])
```

En effet `compression_rectangle(img1)` cherche un plus grand rectangle uni en `(0,3)` avant d'en chercher un en `(2,2)`, donc on couvre le pixel du coin inférieur droit avec le rectangle suivant. `{"pos":(0,3), "coul":1, "ha":3, "la":1}` .

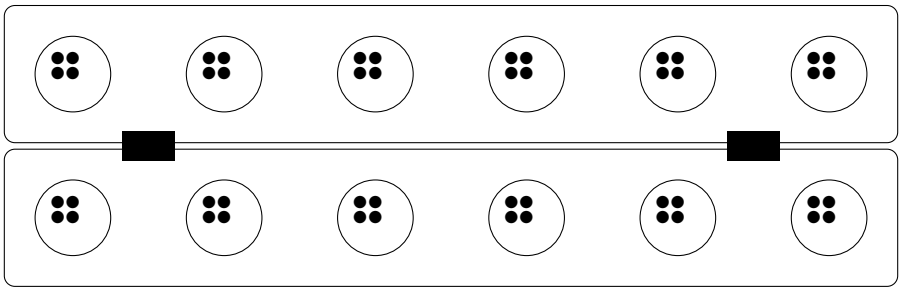
Question 17
Écrire le code d'une telle fonction `compression_rectangle` .

Question 18
Écrire le code d'une telle fonction `decompression_rectangle` .

Exercice 2 : Le jeu de l'Awalé

Introduction

L'awalé est un jeu de société traditionnel africain qui se joue à deux. Le but du jeu est de collecter le plus de graines possible. Il se joue sur un plateau avec 12 cases (6 pour chaque joueur) et 48 graines au total.

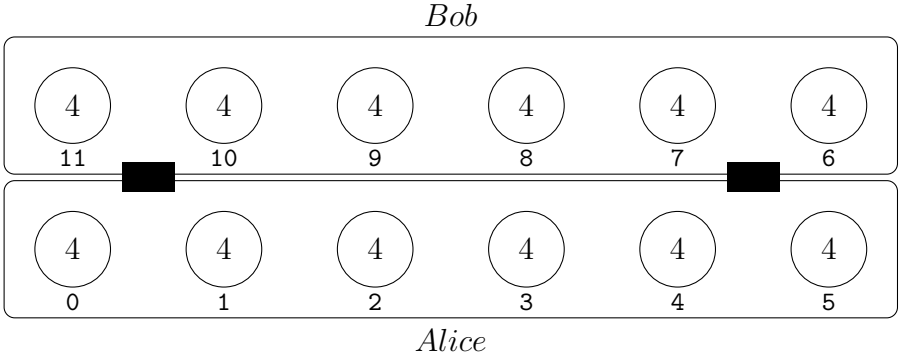


Exemple de plateau en début de partie

Le fonctionnement du jeu est le suivant.

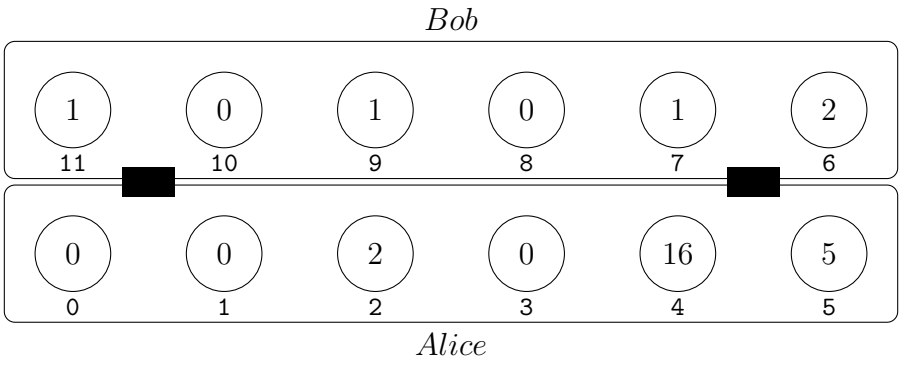
- Chaque joueur possède les 6 cases de son côté du plateau.
- Au début de la partie, les graines sont réparties équitablement dans toutes les cases, il y a donc initialement 4 graines par case.
- Chaque joueur joue à tour de rôle. Le tour d'un joueur consiste en 3 étapes.
  1. Il choisit une case de son côté qui n'est pas vide et prend toutes les graines qui s'y trouvent.
  2. Il les sème une par une dans les cases suivantes, dans le sens inverse des aiguilles d'une montre. On ne sème jamais de graine dans la case où l'on a pris les graines. En particulier, s'il y avait 12 graines ou plus dans la case choisie, il faut effectuer plusieurs fois le tour du plateau pour semer toutes les graines, et **à chaque fois on 'saute' la case d'origine**.
  3. Il récolte des graines dans les cases de l'adversaire selon des règles que nous verrons plus tard dans le sujet.
- Lorsqu'un joueur a récolté plus de la moitié des graines, la partie s'arrête. D'autres cas d'arrêt de la partie seront détaillés plus tard dans le sujet.
- À la fin de la partie, le joueur gagnant est celui qui a récolté le plus de graines.

Pour simplifier la représentation du jeu, on indice les cases de `0` à `11`. Dans les figures représentant un plateau de jeu d'awalé, on représentera les graines par leur nombre et on indiquera de chaque côté du plateau le nom des joueurs. Le joueur courant (c'est-à-dire celui dont c'est le tour de jouer), est toujours le joueur en bas du plateau, et ses cases sont toujours celles d'indice de `0` à `5`.

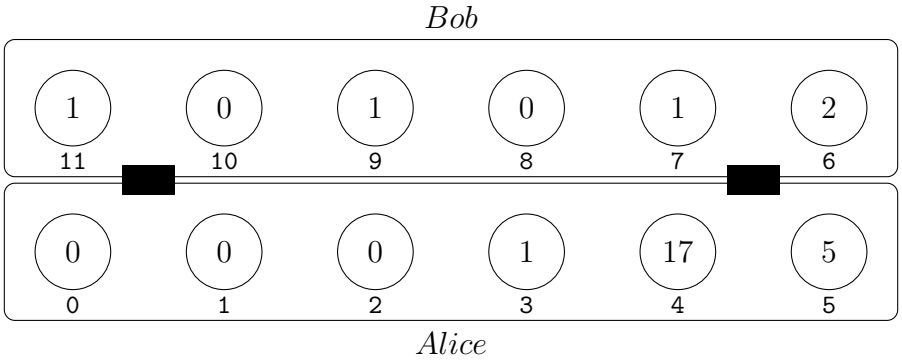


Représentation numérique du plateau au début de la partie

Afin d'illustrer un tour de jeu on considère le plateau suivant.



Sur ce plateau, Alice peut chosir de jouer les cases d'indice 2, 4 ou 5. Si elle joue la case d'indice 2, elle prend les deux graines, puis les sème dans les cases d'indice 3 et 4. On obtient alors le plateau suivant.



Question 1

a) Donner l'état du plateau après qu'Alice a semé les graines dans le cas où elle choisit la case d'indice 4 plutôt que la 2 (on repart du plateau en haut de page où la case 2 était non vide).

b) Même question si elle choisit plutôt la case d'indice 5.

Partie 1 : Semaille

Afin de modéliser une partie d'awalé, on utilise le langage Python. On stocke toutes les données représentant un état d'une partie dans un dictionnaire. La fonction `initialisation` donnée ci-dessous crée un dictionnaire représentant l'état initial de la partie.

```
def initialisation(nom_joueur1, nom_joueur2) :
    '''
    Crée un dictionnaire représentant l'état initial d'une partie dont
    les deux joueurs se nomment nom_joueur1 et nom_joueur2 respectivement
    '''
    etat_jeu = {}
    etat_jeu["joueur1"] = nom_joueur1
    etat_jeu["joueur2"] = nom_joueur2
    etat_jeu["score_j1"] = 0
    etat_jeu["score_j2"] = 0
    etat_jeu["nb_tours"] = 0
    # ci-dessous on crée un tableau contenant 12 valeurs égales à 4
    etat_jeu["plateau"] = [4] * 12
    return etat_jeu
```

Ce dictionnaire contient 6 associations dont les clefs et les valeurs sont les suivantes :

- la valeur associée à la clef `"joueur1"` est une chaîne de caractères désignant le joueur 1 (le joueur 1 est celui qui commence la partie) ;
- la valeur associée à la clef `"joueur2"` est une chaîne de caractères désignant le joueur 2 ;
- la valeur associée à la clef `"score_j1"` est un entier donnant le score du joueur 1 ;
- la valeur associée à la clef `"score_j2"` est un entier donnant le score du joueur 2 ;
- la valeur associée à la clef `"nb_tours"` est un entier donnant le nombre de tours de jeu effectués (cet entier est initialisé à 0 et on lui ajoute 1 à chaque fois qu'un joueur fait un tour) ;
- la valeur associée à la clef `"plateau"` est tableau de 12 entiers qui donnent le nombre de graines contenues dans chacunes des 12 cases du plateau.

Question 2

a) On rappelle qu'au début d'une partie c'est `joueur1` qui commence à jouer. Quelle est la parité de la valeur associée à la clef `"nb_tours"` lorsque c'est au tour de `joueur1` de jouer ?

b) Écrire le code de la fonction `tour_joueur1` qui prend en paramètre un dictionnaire `etat_jeu` contenant les données d'une partie comme décrit précédemment et renvoie `True` si c'est au tour du joueur 1 de jouer et `False` sinon.

Question 3

Écrire le code d'une fonction `somme` qui prend en paramètres un plateau `plt` (c'est-à-dire une liste de 12 entiers) et deux indices `n` et `p` tels que `n` $\leqslant$ `p` et qui renvoie le nombre de graines contenues dans les cases d'indices compris entre `n` inclus et `p` inclus du plateau `plt`.

Retournement de plateau. On rappelle que le plateau est séparé en deux parties de 6 cases. Les six premières cases (celles d'indice de `0` à `5`) sont celles du joueur courant, tandis que les six dernières (celles d'indice de `6` à `11`) sont celles de son adversaire. Afin que ceci reste vrai, il faut, à la fin de chaque tour, échanger les deux parties de six cases, on dit alors qu'on **tourne** le plateau. Par exemple le plateau de la figure 3 ci-après devient le plateau de la figure 4.

Question 4

a) Le tableau correspondant au plateau représenté figure 3 est `[0,0,2,0,16,5,2,1,0,1,0,1]`. Écrire le tableau correspondant au plateau après retournement, c'est-à-dire celui de la figure 4.

b) Écrire le code de la fonction `tourner_plateau` qui prend en paramètre `plt` un plateau (c'est-à-dire une liste de 12 entiers) et qui échange ses cases d'indices `0` à `5` avec celles d'indice `6` à `11` de manière à tourner ce plateau.

Semaille des graines. On s'intéresse maintenant à la semaille des graines. On rappelle que la semaille de graines consiste, pour le joueur courant, à prendre toutes les graines d'une de ses cases non vide et à les semer une à une, dans le sens inverse des aiguilles d'une montre, dans chacune des cases suivantes en sautant la case où il a pris les graines.

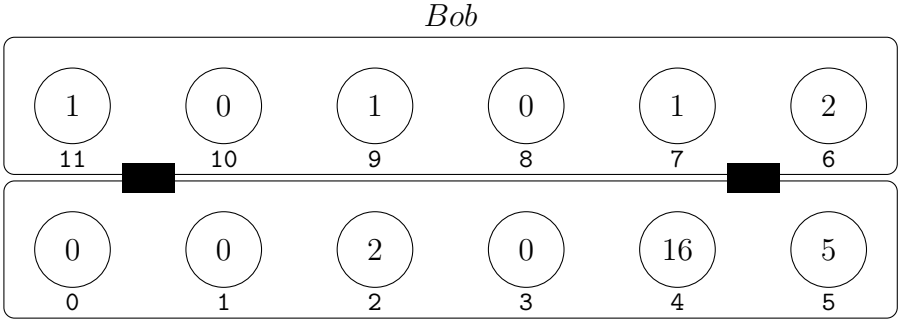


FIGURE 3

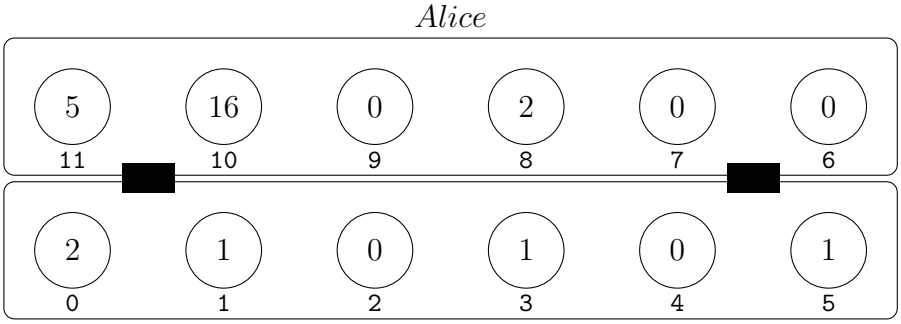


FIGURE 4

Question 5

Écrire une fonction `semer_graines` qui prend en paramètres un plateau de jeu `plt` et l'indice `c` d'une case non vide du plateau et qui :

- modifie en place le plateau `plt` (mise à 0 de la case choisie, ajouts dans les cases où l'on sème),
- et renvoie l'indice de la case où la dernière graine a été semée.

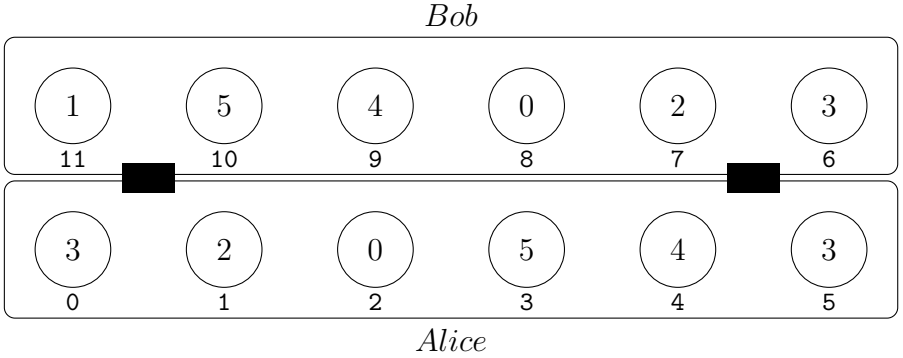
Partie 2 : Récolte

Graines ramassables. Après avoir semé ses graines, le joueur courant peut ramasser des graines, mais seulement dans des cases ramassables. Une case est dite **ramassable** (par le joueur courant) lorsqu'elle vérifie les deux conditions suivantes :

- la case doit appartenir à son adversaire (elle est donc située sur la seconde moitié du plateau) ;
- la case doit contenir exactement 2 ou 3 graines.

Question 6

a) On suppose qu'Alice vient de semer et que le plateau à l'issue de cette semaille est le plateau ci-dessous. Indiquer quelles cases sont alors ramassables par Alice ?

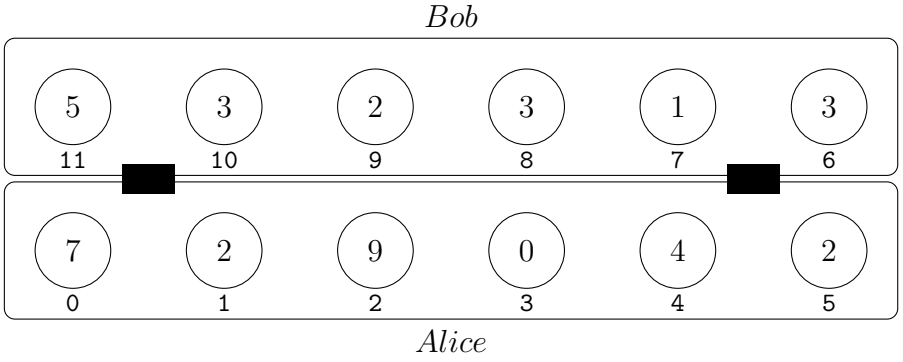


b) Écrire le code de le fonction `case_ramassable` qui prend en paramètres un plateau `plt` et l'indice d'une case `c` et qui renvoie `True` si la case est ramassable, `False` sinon.

Ramassage des graines (ou récolte). Après avoir semé sa dernière graine, le joueur passe à la récolte. Il doit alors ramasser toutes les cases ramassables en commençant par la dernière case semée et en tournant dans le sens des aiguilles d'une montre. Le ramassage s'arrête à la première case non ramassable rencontrée. Il se peut que le ramassage s'arrête avant même d'avoir commencé, par exemple si la dernière case semée par le joueur courant est l'une de ses propres cases.

Question 7

a) On suppose qu'Alice vient de semer les graines de sa case d'indice `3`, qu'elle a semé sa dernière graine dans la case d'indice `10`, et que le plateau à l'issue de cette semaille est le plateau ci-dessous. Combien de graines va-t-elle alors ramasser lors de la récolte ?

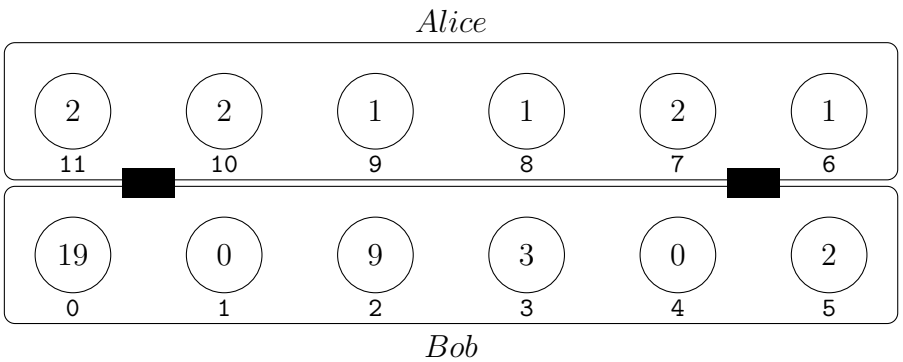


b) Écrire le code de la fonction `ramasser_graines` qui prend en paramètres un plateau de jeu `plt` et l'indice `c` de la dernière case semée et renvoie le nombre de graines ramassées par le joueur courant. Cette fonction utilisera la fonction de la question précédente `case_ramassable` et devra modifier le plateau `plt` (pas de création de nouveau plateau) pour mettre son état à jour après la récolte en plus de renvoyer le nombre de graines ramassées.

Règle de la famine. Cette règle impose au joueur courant de ne pas affamer son adversaire par sa récolte c'est-à-dire qu'après avoir semé et récolté, les cases de son adversaire ne doivent pas toutes être vides.

Question 8

a) Sur le plateau ci-dessous, parmi les cases d'indices `0`, `2` et `5` laquelle entrainera une situation de famine si elle est jouée par Bob ?



b) Écrire le code de la fonction `test_famine` qui prend en paramètres un plateau `plt` et l'indice d'une case `c` et qui renvoie `True` si, en jouant la case d'indice `c` le joueur courant crée une situation de famine pour son adversaire, et `False` sinon. Le plateau `plt` ne devra pas être modifié, pour cela on pourra faire une copie du plateau à l'aide de la fonction `copy`. Par exemple, l'instruction `tab2 = tab1.copy()` permet de copier `tab1` dans une variable `tab2`.

Partie 3 : Programme principal

Cases acceptables. À chaque tour de jeu, il faut vérifier que le joueur peut jouer la case qu'il a choisie. Lorsque le joueur choisit une case, celle-ci est **acceptable** si elle remplit les trois conditions suivantes :

- elle est du côté du joueur courant ;
- elle est non vide ;
- elle ne met pas l'adversaire en famine.

Question 9
Écrire le code de la fonction `test_case` qui prend en paramètres un plateau `plt` et l'indice d'une case `c` et qui renvoie `True` lorsque la case `c` du plateau `plt` est acceptable pour le joueur courant, et `False` sinon.

Question 10
Écrire le code de la fonction `cases_possibles` qui prend en paramètres un dictionnaire `etat_jeu` contenant l'état courant du jeu et qui renvoie la liste des indices des cases acceptables pour le joueur courant. Le dictionnaire `etat_jeu` ne devra pas être modifié.

Règles de fin de partie. Après un tour, il faut tester si la partie est terminée ou si les joueurs peuvent continuer à jouer. Le jeu se termine si l'une des conditions suivantes est vérifiée :

- un des joueurs a ramassé plus de la moitié des graines (soit au moins 25) ;
- les deux joueurs ont ramassé à eux deux plus de 45 graines ;
- le joueur qui va jouer ne possède plus de case acceptable (on suppose que le plateau a été tourné avant de faire les tests, donc les cases du joueur courant sont celles d'indices `0` à `5`) ;
- le nombre de tours joués est supérieur ou égal à 100 (pour éviter un jeu infini).

Question 11
Écrire le code de la fonction `tour_termine` qui prend en paramètre un dictionnaire `etat_jeu` contenant l'état courant du jeu et qui renvoie `True` si le jeu peut continuer, `False` sinon.

Réaliser un tour de jeu. Une ébauche de la fonction `tour_jeu` est donnée ci-après. Cette fonction prend en paramètres, un dictionnaire `etat_jeu` décrivant un état du jeu où la partie n'est pas finie et l'indice `c` d'une case et, si la case `c` est acceptable, elle fait jouer le joueur courant à la case `c` et retourne le plateau puis elle renvoie `True` si la partie peut continuer après ce tour et `False` sinon. Si la case `c` n'est pas acceptable pour le joueur courant, cette fonction affiche un message et renvoie `True` pour permettre au joueur de proposer une nouvelle case.

```
1  def tour_jeu(etat_jeu, c) :
2      '''
3      Modifie etat_jeu en faisant jouer le joueur courant à la case c
4      et renvoie un booléen indiquant si le jeu peut continuer après
5      ce tour, pourvu que la case c soit acceptable.
6      Si la case c n'est pas acceptable, on affiche un message et renvoie
7      True pour permettre au joueur de proposer une nouvelle case.
8      '''
9      plt = etat_jeu["plateau"]
10     # si la case jouée est acceptable :
11     if test_case(plt,c):
12         # Instruction1 : semer les graines
13         ...
14         # Instruction2 : ramasser les graines
15         ...
16         if ... : # Condition1 : c'est j1 qui joue d'après le nombre de tours
17             etat_jeu["score_j1"] = etat_jeu["score_j1"] + graines_gagnees
18         else:
19             etat_jeu["score_j2"] = etat_jeu["score_j2"] + graines_gagnees
20         # Instruction3 : incrémenter le nombre de tours
21         ...
22         tourner_plateau(etat_jeu["plateau"])
23         return tour_termine(etat_jeu)
24     else:
25         print("La case choisie n'est pas acceptable")
26         return True
```

Question 12

Écrire sur la copie l'instruction 1 (ligne 13), l'instruction 2 (ligne 15), la condition 1 (ligne16) et l'instruction 3 (ligne 21) permettant de compléter la fonction `tour_jeu`.

Conditions de victoire. Une fois que la partie est terminée, chaque joueur ramasse les graines restant dans ses 6 cases puis on compare les nombres de graines ramassées par chacun depuis le début de la partie. Un joueur est gagnant s'il a strictement plus de graines que l'autre.

Question 13

Écrire le code de la fonction `gagnant` qui prend en paramètre un dictionnaire `etat_jeu` contenant l'état courant du jeu. Cette fonction doit procéder au dernier ramassage des graines de chacun des deux joueurs, les ajouter à leurs scores, puis renvoyer le nom du joueur gagnant, c'est-à-dire le nom de celui qui a le plus de graines à la fin du jeu ou, en cas d'égalité, la chaîne de caractères `"égalité"`.

Question 14

Écrire le code de la fonction `awale` qui propose aux utilisateurs de jouer une partie d'awalé. Cette fonction prend en paramètres deux chaînes de caractères `nom_joueur1` et `nom_joueur2` pour nommer les joueurs. À chaque tour, la fonction teste si la partie peut continuer :

- si oui, elle affiche le plateau et demande au joueur courant de choisir une case ;
- si la partie est terminée, elle affiche le nom du gagnant (ou `"égalité"` le cas échéant).