Banque « Agro- Veto »
AT - 0518

Algorithmique et Informatique
Durée : 45 mn

Si au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.
L'usage d'une calculatrice est interdit pour cette épreuve.

- On suppose dans tout le sujet que les modules `random` et `numpy` ont été préalablement importés via les instructions `import random as rd` et `import numpy as np`.
- La fonction `rd.choice` du module `random` renvoie un élément choisi aléatoirement dans une séquence.

  Par exemple, `rd.choice([4,6,2,3])` *peut renvoyer 6.*
- La fonction `np.array` du module `numpy` définit le type `array` (tableau) par conversion de liste de listes, permettant de représenter des matrices.

  Par exemple, `mat = np.array([[1,2,3],[4,5,6]])` *représente la matrice* $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$.
- On accède à un élément d'une matrice `mat` ayant pour numéro de ligne `i` et numéro de colonne `j` (la numérotation commençant à 0) par la syntaxe : `mat[i,j]`.

  En reprenant l'exemple ci-dessus, `mat[0,0]`, `mat[1,0]`, `mat[0,1]` *et* `mat[1,1]` *valent respectivement 1, 4, 2 et 5.*
- la fonction `np.shape` renvoie le couple (n, p) où n et p désignent respectivement le nombre de lignes et de colonnes d'une matrice passée en argument.

  On récupère leur valeur par l'instruction : `n,p = np.shape(mat)`.

  Avec les notations de l'exemple ci-dessus, `n` *et* `p` *prendront respectivement les valeurs 2 et 3.*

Modélisation d'un labyrinthe parfait

- On modélise un labyrinthe par une matrice dont chaque coefficient est égal à :
  - 0 si la case est vide (et ne correspond ni à l'entrée ni à la sortie du labyrinthe) ;
  - 1 si la case est un mur ;
  - 2 si la case (vide) correspondante est l'entrée du labyrinthe ;
  - 3 si la case (vide) correspondante est la sortie du labyrinthe.

  Les bords extérieurs du labyrinthe ne font pas partie de la modélisation.
- On appelle **labyrinthe parfait** tout labyrinthe vérifiant que, pour tout couple de *cases vides*, il existe un unique chemin qui les relie (sans passer deux fois par le même point).

L'objectif de ce problème est de déterminer le **chemin-solution** reliant l'entrée et la sortie d'un labyrinthe parfait. On représente un labyrinthe parfait en figure 1 et sa modélisation par une matrice (de type `array`) ci-dessous.

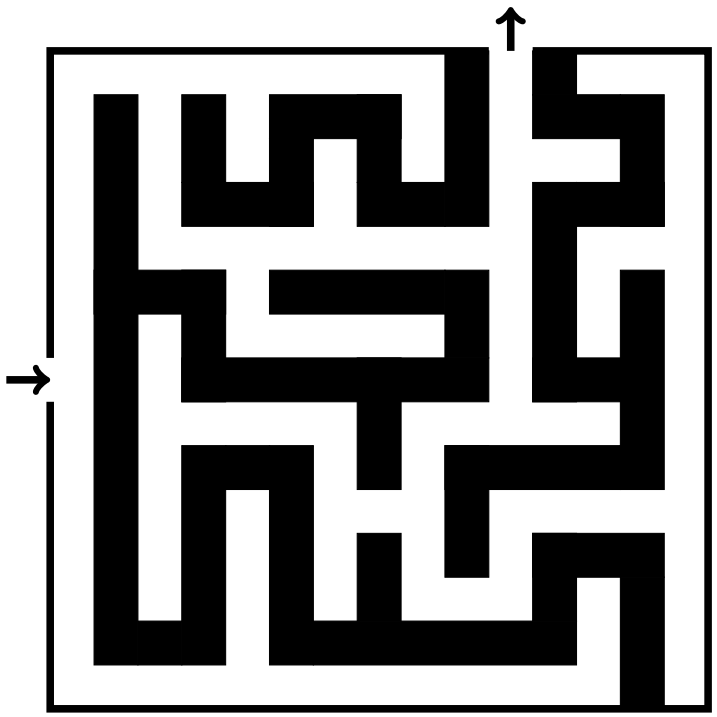

Figure 1 – exemple de labyrinthe parfait

```
labyrinthe = np.array([
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 3, 1, 0, 0, 0],
[0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0],
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0],
[0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0],
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
[0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0],
[0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0],
[2, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0],
[0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0],
[0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0],
[0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0],
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0],
[0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0],
[0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
])
```

Code 1 – modélisation du labyrinthe représenté en figure 1

1 Préliminaires

On considère le labyrinthe représenté sur la figure 2 ci-dessous.

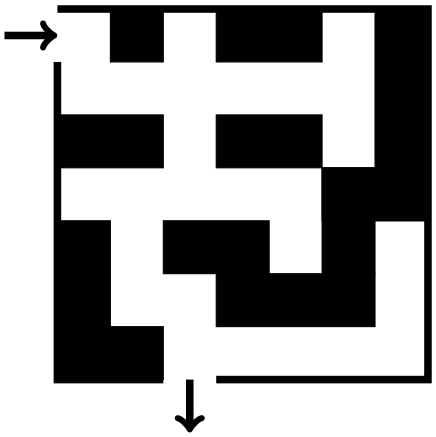

Figure 2 – exemple de labyrinthe parfait

a. Parmi les quatre matrices ci-dessous, indiquer le nom de l'unique `array` qui modélise le labyrinthe représenté sur la figure 2.

```
maze1 = np.array([
[0, 1, 0, 1, 1, 0, 1],
[0, 0, 0, 0, 0, 0, 1],
[1, 1, 0, 1, 1, 0, 1],
[0, 0, 0, 0, 0, 1, 1],
[1, 0, 1, 1, 0, 1, 0],
[1, 0, 0, 1, 1, 1, 0],
[1, 1, 0, 0, 0, 0, 0]
])
```

```
maze2 = np.array([
[3, 1, 0, 1, 1, 0, 1],
[0, 0, 0, 0, 0, 0, 1],
[1, 1, 0, 1, 1, 0, 1],
[0, 0, 0, 0, 0, 1, 1],
[1, 0, 1, 1, 0, 1, 0],
[1, 0, 0, 1, 1, 1, 0],
[1, 1, 0, 0, 0, 0, 2]
])
```

```
maze3 = np.array([
[2, 1, 0, 1, 1, 0, 1],
[0, 0, 0, 0, 0, 0, 1],
[1, 1, 0, 1, 1, 0, 1],
[0, 0, 0, 0, 0, 1, 1],
[1, 0, 1, 1, 0, 1, 0],
[1, 0, 0, 1, 1, 1, 0],
[1, 1, 0, 0, 0, 0, 3]
])
```

```
maze4 = np.array([
[2, 1, 0, 1, 1, 0, 1],
[0, 0, 0, 0, 0, 0, 1],
[1, 1, 1, 1, 1, 0, 1],
[0, 0, 1, 0, 0, 1, 1],
[1, 0, 1, 1, 0, 1, 0],
[1, 0, 1, 1, 1, 1, 0],
[1, 1, 3, 0, 0, 0, 0]
])
```

b. On modélise le chemin-solution d'un labyrinthe parfait par une liste de listes $[i, j]$ où i et j désignent respectivement le numéro de ligne et de colonne de chaque case du chemin, depuis l'entrée vers la sortie.

Parmi les quatre listes ci-dessous, indiquer l'unique liste modélisant le chemin-solution du labyrinthe représenté sur la figure 2.

```
chemin1=[[0,0],[1,0],[1,1],[1,2],[2,2],[3,2],[3,1],[4,1],[5,1],[5,2],[6,2]]
chemin2=[[6,2],[5,2],[5,1],[4,1],[3,1],[3,2],[2,2],[1,2],[1,1],[1,0],[0,0]]
chemin3=[[0,0],[0,1],[1,1],[2,1],[2,3],[1,3],[1,5],[2,5],[2,6]]
chemin4=[[0,0],[0,1],[1,1],[2,1],[2,2],[2,3],[1,3],[1,4],[1,5],[2,5],[2,6]]
```

c. Écrire une fonction `entreeSortie` qui prend en argument une matrice modélisant un labyrinthe et qui renvoie une liste de la forme `[[i0,j0], [i1,j1]]` où `i0` et `j0` désignent respectivement le numéro de ligne et de colonne de l'entrée, et `i1` et `j1` ceux de la sortie.

2 Recherche de chemin dans un labyrinthe parfait

On propose d'utiliser la méthode du retour sur trace pour déterminer le chemin-solution qui relie l'entrée et la sortie. Son principe est de parcourir le labyrinthe en marquant les cases visitées. L'algorithme est décrit ci-dessous.

- Initialement, on se trouve à l'entrée du labyrinthe.
  - On initialise le chemin-solution à la liste contenant uniquement la liste `[i_0,j_0]` des numéros de ligne et de colonne de l'entrée du labyrinthe ;
  - on marque cette case comme visitée en modifiant la valeur stockée dans la matrice. On marquera les cases visitées par l'entier 4 : `labyrinthe[i_0, j_0] = 4` ;
  - on note i et j les numéros de ligne et de colonne de la case où l'on se trouve actuellement.
- Tant que la case actuelle n'est pas la sortie :
  - on détermine, parmi les cases adjacentes (dont le nombre est au plus égal à 4) de la case actuelle, celles qui n'ont pas été visitées, i.e. celles non marquées par un 1 (mur) ou un 4 (case visitée) ;
  - si la liste des cases adjacentes non visitées est non vide, on en choisit une au hasard qui devient la nouvelle case actuelle ; on marque cette case par un 4 dans la matrice et on l'ajoute au chemin en construction ;
  - si la liste des cases adjacentes non visitées est vide, c'est qu'on se trouve dans une *impasse* ; on rebrousse alors chemin en supprimant du chemin la dernière case insérée. La nouvelle case actuelle devient alors la dernière case du chemin modifié.

a. Recopier et modifier le code de la fonction `vide` ci-dessous de manière à ce que `vide(lab, x, y)` renvoie `True` si, et seulement si, `[x,y]` correspond aux indices d'une case valide du labyrinthe `lab` et non encore visitée.

On rappelle que l'entrée et la sortie sont des cases vides.

```
def vide(lab, x, y):
    n, p = np.shape(lab)
    if 0< x <= p and 0 <= y < n and lab[x,y] not in [0,2]:
        return False
    else:
        return True
```

b. Recopier et compléter sur votre copie la fonction ci-dessous de manière à ce qu'elle renvoie la liste des cases adjacentes non visitées d'une case d'indices $[i, j]$ d'un labyrinthe `lab`.

```
def voisinsNonVisites(lab, i, j):
    liste = ......
    cases = [[i+1,j], [i-1,j], [i,j-1], [i,j+1]]
    for k in range(......):
        x = cases[k][0]
        y = cases[k][1]
        if ......:
            liste.append([x,y])
    return liste
```

c. Expliquer, lors de la construction de la liste `chemin` représentant le chemin-solution d'un labyrinthe, comment supprimer, en cas d'impasse, la dernière case insérée.

d. Recopier et compléter sur votre copie le code ci-après afin de définir une fonction `chemin` prenant en argument une matrice (de type `array`) modélisant un labyrinthe et qui renvoie la liste des indices $[i, j]$ (numéros de ligne et colonnes) des cases du chemin reliant l'entrée et la sortie du labyrinthe.

```
def cheminSolution(lab):
    entree, sortie = ......
    i, j = entree
    chemin = [......]
    lab[i,j] = ......
    while [i, j] != ...... :
        liste = voisinsNonVisites(......, ......, .......)
        if len(liste) > ......:
            i, j = rd.choice(......)
            lab[i,j] = ......
            chemin.append(......)
        else:
            ......
            i, j = chemin[......]
    return ......
```

3 Compression de chemin

La liste représentant le chemin-solution est parfois très longue. On souhaite la compresser en ne mémorisant que les cases où l'on change de direction.
Le principe est de retenir la dernière direction suivie et de la comparer avec la direction actuelle sous la forme d'un "vecteur déplacement" $[i_{k+1}-i_k, j_{k+1}-j_k]$ où $[i_k, j_k]$ et $[i_{k+1}, j_{k+1}]$ désignent les indices de deux cases consécutives du chemin-solution.
Ainsi, le chemin compressé du labyrinthe représenté sur la figure 2 est :

```
[[0, 0], [1, 0], [1, 2], [3, 2], [3, 1], [5,1], [5,2], [6,2]]
```

a. Quel est le chemin compressé du labyrinthe représenté sur la figure 1 ?

b. Écrire une fonction `direction` qui prend en argument une liste représentant le chemin-solution d'un labyrinthe parfait et un entier `k` et qui renvoie la liste $[i_{k+1} - i_k, j_{k+1} - j_k]$ où $[i_k, j_k]$ et $[i_{k+1}, j_{k+1}]$ sont respectivement les indices des k-ème et $(k+1)$-ème cases de la liste `chemin`.

On ne vérifiera pas ici que les entiers k et $k+1$ sont des indices valables de la liste `chemin`.

c. Recopier et compléter sur votre copie le code de la fonction `compression` de manière à ce que l'instruction `compression(chemin)` renvoie une liste (de listes de deux entiers) représentant le chemin compressé de la liste `chemin` passée en argument et représentant un chemin-solution.

On supposera toujours qu'un chemin-solution est de longueur supérieure ou égale à 2.

```
def compression(chemin):
    dir = direction(chemin, ......)
    chemin_compresse = [chemin[0]]
    for k in range(......, ......):
        if dir != ......:
            chemin_compresse.append(......)
            dir = direction(chemin, k)
    chemin_compresse.append(......)
    return chemin_compresse
```

FIN DU SUJET