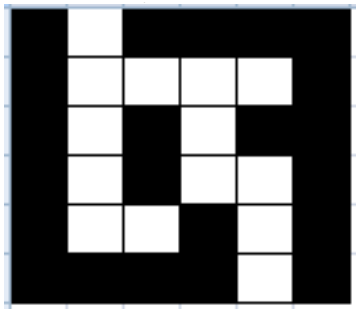
Structures abstraites 1
TP : Résolution de labyrinthes

Les piles permettent de résoudre un grand nombre de problèmes sur grilles comme le Démineur. Dans ce TP, nous allons étudier le problème classique de la résolution de labyrinthes à l'aide de piles.

On représentera le labyrinthe à l'aide d'un tableau bidimensionnel (en Python : une liste de listes) où l'on ne peut se déplacer qu'à gauche, à droite, en haut ou en bas.

Le labyrinthe que nous allons étudier est dessiné ci-dessous, avec les murs en noir.



```
laby = [[0,1,0,0,0,0],
        [0,1,1,1,1,0],
        [0,1,0,1,0,0],
        [0,1,0,1,1,0],
        [0,1,1,0,1,0],
        [0,0,0,0,1,0],]
```

En Python, il est codé par le tableau de droite, avec la valeur 0 pour les murs :

- on repère une case par ses coordonnées `(i, j)`, on y accède dans le tableau par `laby[i][j]` ;
- l'entrée se fait par la case (0,1) et la sortie par la case (5,4).

1) Préliminaires

Dans cette section, nous allons aborder quelques notions fondamentales à la réalisation de notre programme.

Question 1 :

Compléter le code pour obtenir le nombre de lignes et le nombre de colonnes du tableau `laby` :

```
lignes = len(......)
colonnes = len(......)
```

Question 2 :

Nous aurons besoin de créer une copie du tableau initial `laby` afin de travailler dessus sans modifier le tableau initial `laby`.

- À partir du tableau `laby`, créez un nouveau tableau que l'on appellera `labyCopie`.
- Donnez ensuite la valeur "88" à la ligne 3 et la colonne 2 de votre tableau `labyCopie`.
- Affichez finalement les valeurs de `laby` et `labyCopie` grâce au code page suivante.

```
for ligne in laby:
    print(ligne)
print("---------------")
for ligne in labyCopie:
    print(ligne)
```

- Que remarquez-vous ? En utilisant le cours sur les listes chainées, essayez de fournir une explication à ce phénomène.

Pour éviter ce problème, nous allons utiliser une copie "profonde" du tableau `laby` grâce à la fonction `deepcopy` de la bibliothèque `copy`.
Une fois la bibliothèque importée, on procède à une telle copie à l'aide de la syntaxe :

```
labyCopie = deepcopy(laby)
```

- Modifiez votre programme afin d'effectuer une copie profonde, puis affichez les deux tableaux afin de vérifier le bon fonctionnement.

Question 3 :
Créez une fonction `afficheLaby` qui prend en argument un tableau et l'affiche ligne par ligne.

2) Fonction voisins

Voici une fonction qui prend en paramètre un tableau T et un tuple v ainsi que quelques jeux de tests :

```
def voisins(T, v):
    V = []
    i, j = v[0], v[1]
    for a in (-1,1):
        if 0 <= i+a < lignes and T[i+a][j] == 1:
            V.append((i+a, j))
        if 0 <= j+a < colonnes and T[i][j+a] == 1:
            V.append((i, j+a))
    return V
```

```
# Jeu de tests.
# Si une AssertionError s'affiche, votre fonction ne passe
# pas les tests et doit être modifiée !
assert(voisins(laby, [2,2]) == [(1, 2), (2, 1), (2, 3)])
assert(voisins(laby, [5,0]) == [])
assert(voisins(laby, [5,5]) == [(5,4)])
```

Question 1 :
Que retourne la fonction `voisins` ?

Question 2 :
Expliquer précisément l'affichage provoqué par l'instruction : `print(voisins(laby,(1,1))` .

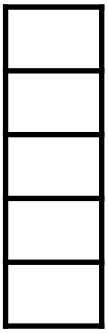
Question 3 :
Spécifier la fonction à l'aide de triple guillemets.

3) Parcours du labyrinthe

L'idée de l'algorithme est de parcourir le labyrinthe depuis l'entrée, en utilisant **une pile pour stocker le chemin courant.** Lorsque le chemin n'aboutit pas, la structure pile permet de dépiler jusqu'à ce qu'une autre voie soit possible. On redémarre la pile sur cette autre voie et on recommence l'algorithme.

Pour plus de clarté, résumons cet algorithme dans un schéma :

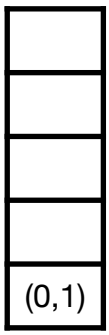
0	1	0	0	0	0
0	1	1	1	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	1	0
0	0	0	0	1	0



Pile

On dispose d'une copie de notre tableau et d'une pile vide.

0	-1	0	0	0	0
0	1	1	1	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	1	0
0	0	0	0	1	0



Pile

On examine la première case (0,1). Cette case est visitée et on la marque en passant sa valeur à -1.

On dresse la liste de ses voisines accessibles : [(1,1)] . Comme cette liste n'est pas vide, on empile la case examinée (0,1) et on examine la première voisine (1,1).

Le processus se poursuit jusqu'à tomber sur une impasse.

0	-1	0	0	0	0
0	-1	1	1	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	1	0
0	0	0	0	1	0

(1,1)
(0,1)

Pile

On examine la case (1,1). Cette case est visitée et on la marque en passant sa valeur à -1.

On dresse la liste de ses voisines accessibles : [(2,1), (1,2)] . Comme cette liste n'est pas vide, on empile la case examinée (1,1) et on examine la première voisine (2,1).

0	-1	0	0	0	0
0	-1	1	1	1	0
0	-1	0	1	0	0
0	-1	0	1	1	0
0	-1	-1	0	1	0
0	0	0	0	1	0

~~(4,1)~~
~~(3,1)~~
~~(2,1)~~
(1,1)
(0,1)

Pile

On examine la case (4,2). Cette case est visitée et on la marque en passant sa valeur à -1.

On dresse la liste de ses voisines accessibles non visitées : [ ] . Comme cette liste est vide, on dépile et on revient à (4,1).

On dresse la liste de ses voisines accessibles non visitées : [ ] etc.

0	-1	0	0	0	0
0	-1	-1	1	1	0
0	-1	0	1	0	0
0	-1	0	1	1	0
0	-1	-1	0	1	0
0	0	0	0	1	0

(1,2)
(1,1)
(0,1)

Pile

On examine la case (1,1). Cette case est visitée et on la marque en passant sa valeur à -1 (c'est d'ailleurs déjà fait…).

On dresse la liste de ses voisines accessibles non visitées : [(1,2)] . Comme cette liste n'est pas vide, on empile la case examinée (1,1) et on examine la première voisine (1,3).

On continue ainsi jusqu'à sortir du labyrinthe.

Question 1 :

À partir de la description détaillée du code, compléter l'algorithme page suivante :

Question 2 :

Quel cas correspond à "la pile est vide" ?

Question 3 :

Écrire ce programme en Python et le tester sur différents labyrinthes (carrés ou non, avec des sorties ou non).

```
fonction parcours( laby , entree, sortie)
    T ← une copie de ........
    p ← Pile()
    v ← entree
    on met à ........ la valeur de la case ........ dans ........
    recherche ← True
    tant que recherche est vrai
        vois ← voisins (........, ........)
        si la liste ......... est vide
            si la pile vide
                renvoyer False
            sinon v ← ........................
        sinon
            on ........ p avec .........
            v ← le 1er ...................
            on met à ..... la valeur de la case ....... dans .......
            si v = ........
                  recherche ← ........
    return ........
```