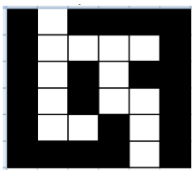
Projet n°3 : Utiliser une pile ou une file pour résoudre un problème

Cette fiche d'exercices s'inspire en partie des documents de Van Zuijlen.

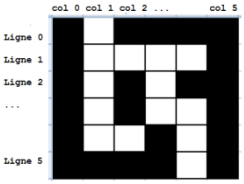
I. Présentation du problème

Dans ce problème, on définit un labyrinthe comme un ensemble de cases pouvant être noires ou blanches. Les cases noires sont des murs et donc infranchissables. Les cases blanches peuvent être franchies.

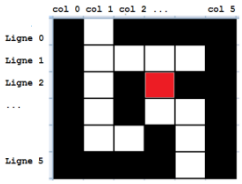
Voici un exemple de labyrinthe contenant 36 cases au total.



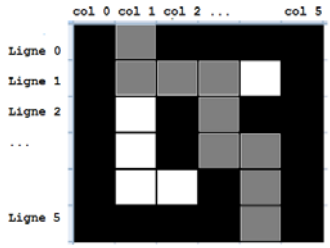
Les lignes sont numérotées à partir de 0 et les colonnes également. Par exemple pour le labyrinthe précédent, les lignes sont numérotées de 0 à 5 et les colonnes de 0 à 5 comme dans l'image ci-contre.



Chaque case est repérée par le numéro de sa ligne et le numéro de sa colonne. Par exemple la case rouge dans la figure ci-dessous a pour coordonnées (2,3) (ligne n°2 et colonne n°3).



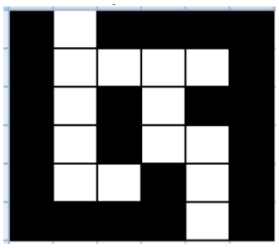
Le but est d'écrire un programme en Python permettant de résoudre un labyrinthe, c'est-à-dire, connaissant la case d'entrée et la case de sortie, déterminer s'il existe un chemin pour arriver à la sortie (pas forcément le meilleur chemin) en se déplaçant vers le haut, le bas, la gauche ou la droite (mais pas en diagonale).
Par exemple, si on reprend toujours le même labyrinthe et on considère que l'entrée est la case de coordonnées (0,1) et la sortie la case de coordonnées (5,4), alors un chemin possible est le chemin :
(0,1) puis (1,1) puis (1,2) puis (1,3) puis (2,3) puis (3,3) puis (3,4) puis (4,4) puis (5,4), ce qui correspond au schéma ci-contre.



II. Représentation d'un labyrinthe en Python

Pour représenter un labyrinthe, on utilisera une liste de listes. Ces listes contiendront des 0 et des 1 : 0 pour une case noire (un mur) et 1 pour une case blanche. La première liste représentera le première ligne du labyrinthe, la deuxième liste la deuxième ligne etc …

Par exemple, si on reprend toujours notre labyrinthe, on peut le représenter par la variable `laby` suivante :



```
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]]
```

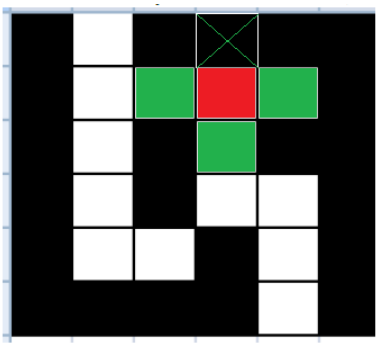
Ainsi, si on veut par exemple savoir si la case de coordonnées (2,3) est un mur ou bien franchissable, il suffit de regarder son coefficient et on y accède par l'instruction `laby[2][3]`

Pour retrouver le nombre de lignes, il suffit de calculer `len(laby)` et pour calculer le nombre de colonnes, il suffit de compter le nombre d'éléments de la première liste par exemple avec `len(laby[0])`.

III. Une fonction intermédiaire

On aura besoin d'une fonction `voisins(lab,case)` qui prend en paramètre un labyrinthe `lab` (sous la forme d'une liste de liste) et les coordonnées d'une case sous la forme d'un tuple, par exemple la case de coordonnées (0,1) est représentée par le tuple (0,1). Cette fonction devra renvoyer les voisins accessibles de la case, c'est à dire parmi les cases voisines, celles qui sont franchissables.

Par exemple, si on prend la case de coordonnées (1,3) (case représentée en rouge), alors cette case possède 3 voisins accessibles : les cases de coordonnées (1,2), (2,3), (1,4), (cases représentées en vert). En revanche, la case de coordonnées (0,3) n'est pas un voisin accessible puisque c'est un mur donc elle ne fait pas partie de la liste renvoyée.



Concrètement, toujours avec notre variable `laby`, voici ce que doit renvoyer `voisins(laby, (1,3))`.

```
>>> 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]]
>>> voisins(laby, (1,3))
[(1, 2), (2, 3), (1, 4)]
```

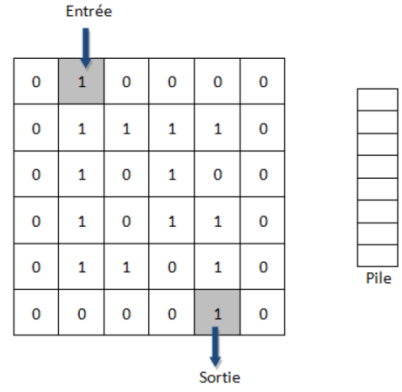
Attention, dans la programmation de la fonction `voisins`, il faut faire attention aux cases situées dans les bords, elles n'ont pas 4 voisins. Si vous bloquez trop pour cette partie, partez du principe que vous avez réussi et essayer de faire la partie suivante.
Ecrire cette fonction `voisins`.

IV. Utiliser une pile pour la fonction principale

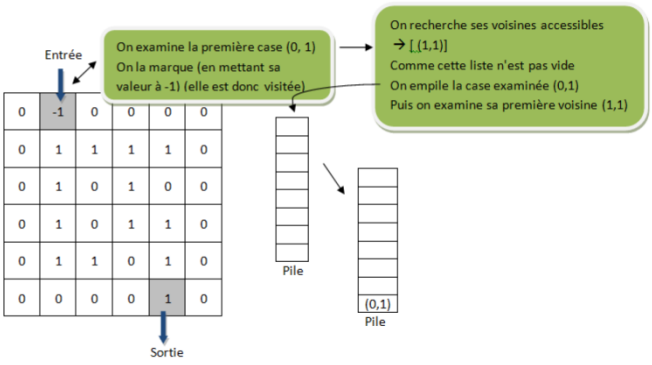
On veut écrire une fonction `resolution(lab, entree, sortie)` qui prend en paramètre un labyrinthe (liste de listes), les coordonnées de la case d'entrée sous la forme d'un tuple, les coordonnées de la case de sortie sous la forme d'un tuple et qui renvoie un chemin sous la forme d'un ensemble de cases comme on a pu le faire à la fin de la partie 1.
Toujours avec le même labyrinthe, en considérant que l'entrée est la case de coordonnées (0,1), que la sortie est la case de coordonnées (5,4), alors on obtient par exemple le chemin suivant (à lire de bas en haut).

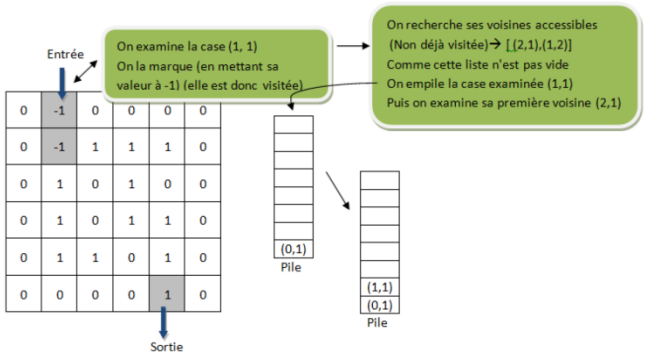
```
>>> print(resolution(laby, (0,1), (5,4)))
(5, 4)
(4, 4)
(3, 4)
(3, 3)
(2, 3)
(1, 3)
(1, 2)
(1, 1)
(0, 1)
```

L'idée est de parcourir le labyrinthe depuis l'entrée, en utilisant une pile pour stocker le chemin, pour pouvoir dépiler lorsque le chemin n'aboutit pas et redémarrer sur une autre voie.
Voici plusieurs schémas explicatifs :



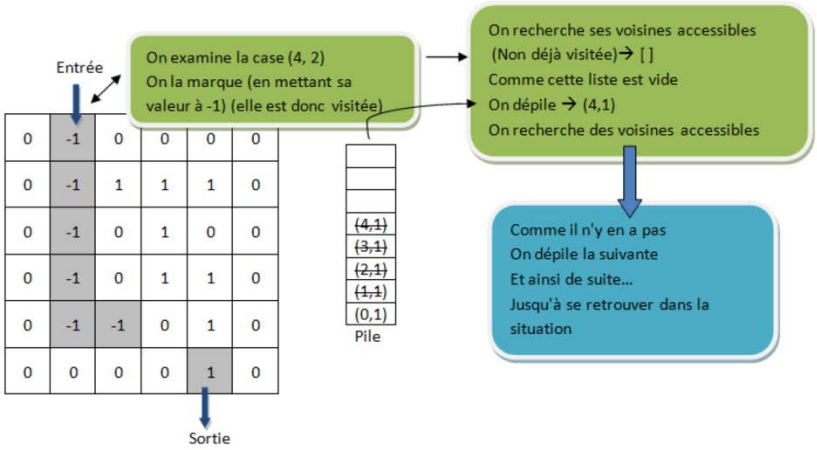
Au départ, on crée une pile.



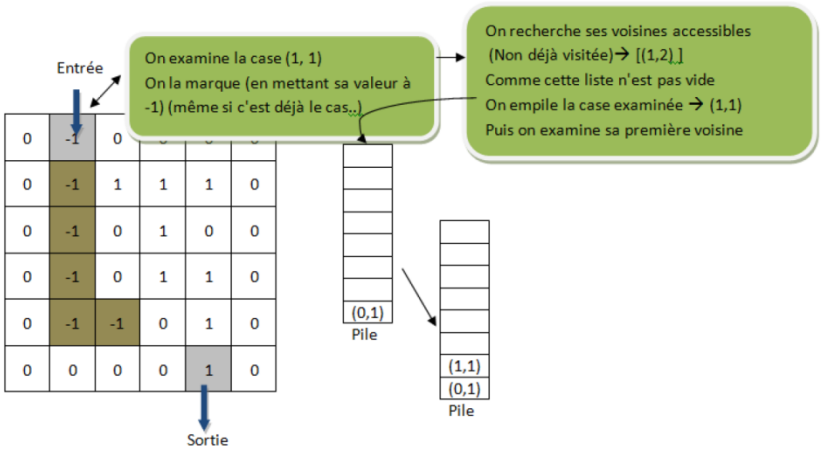


Et ainsi de suite jusqu'à ce que l'on tombe sur une impasse..

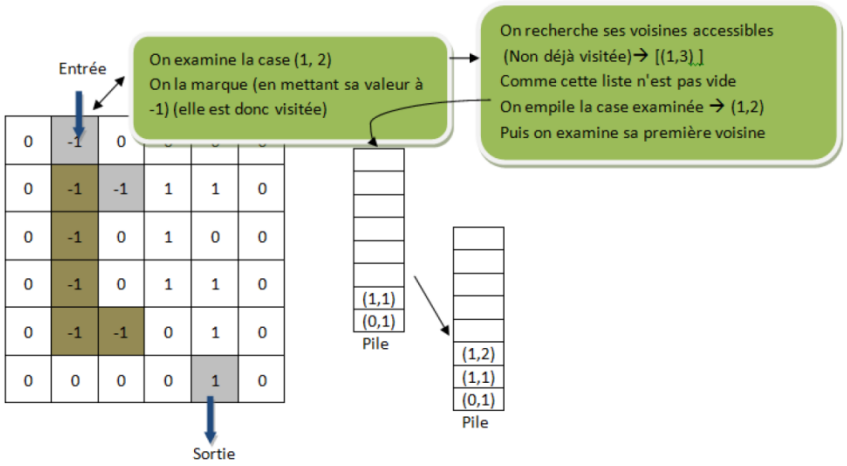
.....



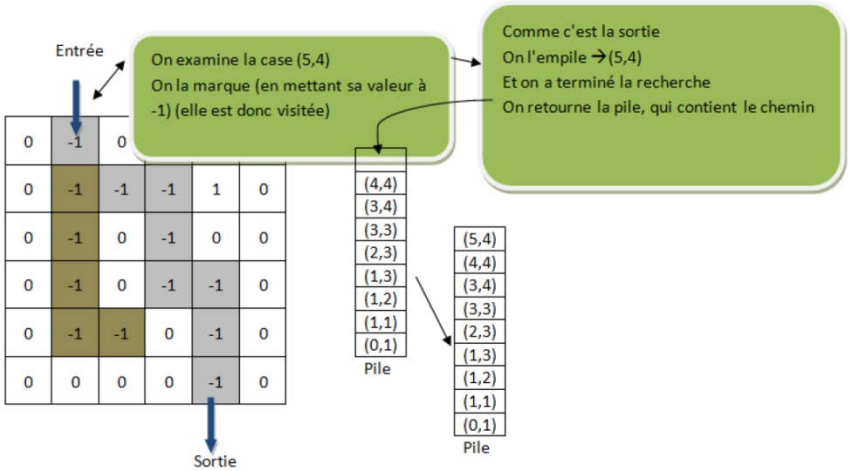
On examine la dernière case dépilée...



On poursuit l'exploration...



Et ainsi de suite jusqu'à la sortie...



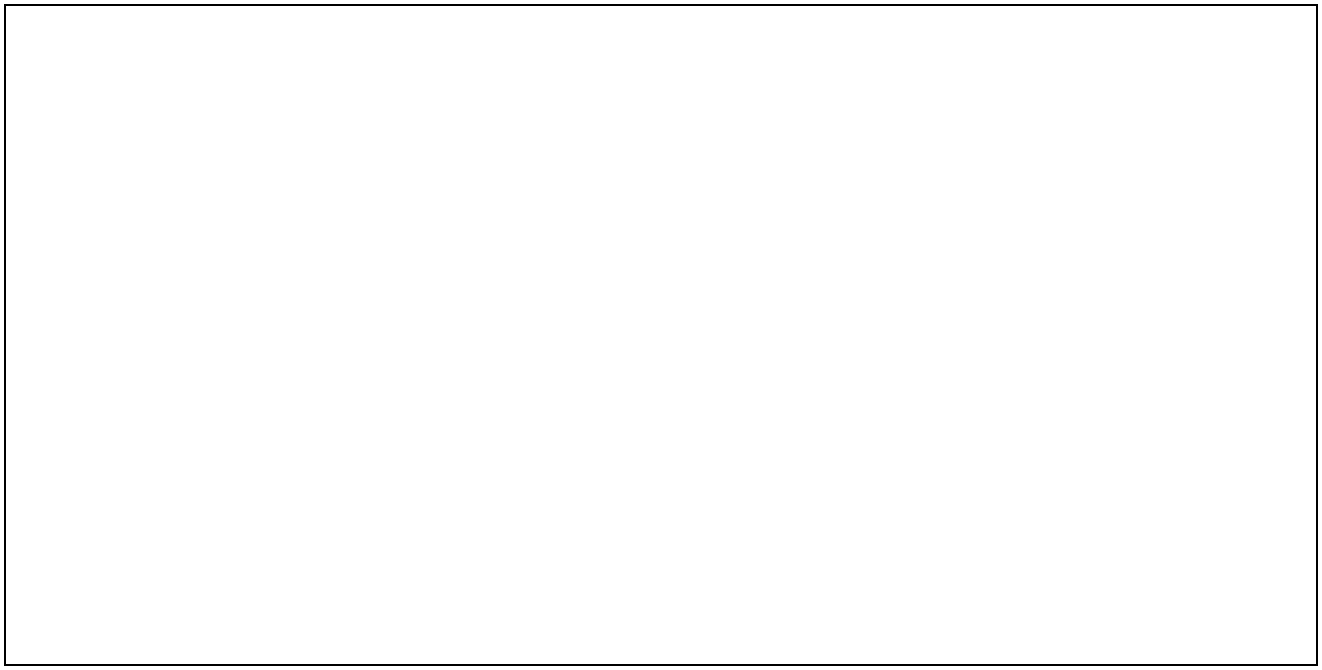
Comme le labyrinthe est modifié lors du parcours (on transforme des coefficients en -1), il est plus prudent de faire une copie du labyrinthe d'entrée grâce à la fonction `deepcopy` du module `copy`.

Voici en partie la fonction `resolution` à compléter. Des commentaires sont là pour vous aider à programmer les grandes étapes. Vous êtes libres d'utiliser cette aide ou pas.

Tout au long du programme, on va créer une variable `case` (un tuple) qui correspondra à la case qui est en cours de visite. Au départ, cette case correspond à la case d'entrée

```
from copy import deepcopy

def resolution(lab, entree, sortie):
    T = deepcopy(lab)  # on copie lab dans une variable T, on travaille donc avec T désormais
    case = entree # au départ, la case en cours de visite est la case d'entrée

    # la case d'entrée a été visitée, il faut modifier le bon coefficient de T en – 1
```

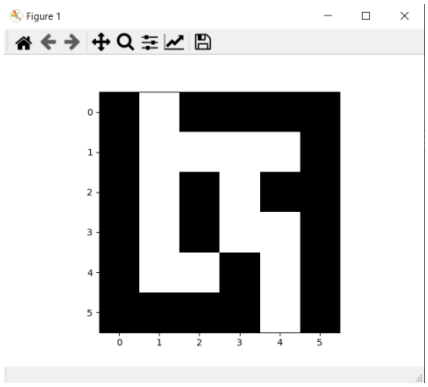
```
    pile = Pile()# création d'une pile vide
    vois = voisins(T, case) # on stocke dans une variable vois les voisins de la case en cours de visite
    continuer = True # variable qui permettra de sortir de la boucle, pour sortir, il suffit de la mettre à False

   while continuer:  # on rentre dans une boucle


        # si vois est vide, on recommence avec la case extraite de la pile
        # sinon, on ajoute la case dans la pile et on recommence avec comme case suivante la première voisine
       # a tout instant, si la pile est vide, c'est que manifestement, il n'y a aucun chemin qui mène à la sortie, on peut sortir de la boucle et renvoyer que c'est impossible. Si la sortie se trouve dans la liste des voisins, on peut sortir de la boucle et renvoyer le chemin obtenu qui correspond à la pile.
```



V. Aller plus loin en représentant graphiquement les labyrinthes pas si compliqué que ça, le plus dur a été fait !)

Ecrivons d'abord une fonction qui prend en paramètre un labyrinthe sous forme d'une liste de listes et qui l'affiche grâce au module `matplotlib.pyplot`.

```
>>> 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]]
>>> affiche_laby(laby)
```

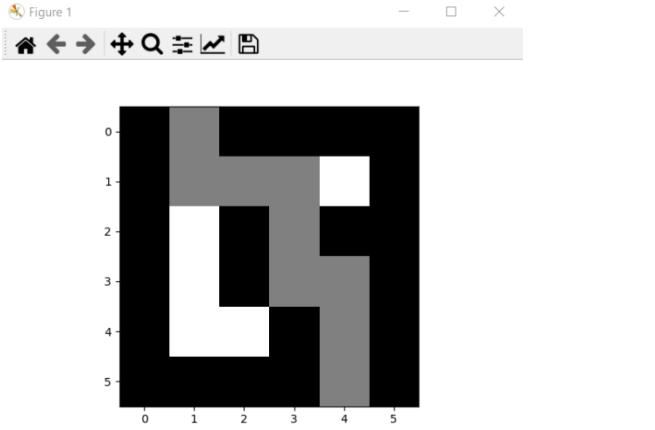


Ce n'est difficile, il suffit d'utiliser le fonction `imshow` de `matplotlib.pyplot` et indiquer un niveau de gris : les 0 sont affichés en noir, les 1 en blanc et toutes valeurs intermédiaires en gris (gris foncé pour les valeurs proches de 0 et gris clair pour les valeurs proches de 1).

```
import matplotlib.pyplot as plt

def affiche_laby(lab):
    plt.imshow(laby, cmap=cm.gray)
    plt.show()
```

Ecrivons désormais une fonction `affiche_solution(lab, entree, sortie)` qui affiche le chemin trouvé en gris (ça sera plus simple).

```
>>> affiche_solution(laby, (0,1), (5,4))
```



On trouve d'abord toutes les cases faisant partie du chemin grâce à la fonction `resolution`.
Ensuite, on peut faire une copie de la liste de listes `lab`, parcourir les coefficients et modifier les cases faisant partie du chemin en leur affectant le coefficient 0.5 (pour qu'elles puisse s'afficher en gris). Enfin, on utilise `matplotlib.pyplot`.

On peut aller plus loin en générant des labyrinthes de taille arbitraire puis en les résolvant…