TP 6: Labyrinthe

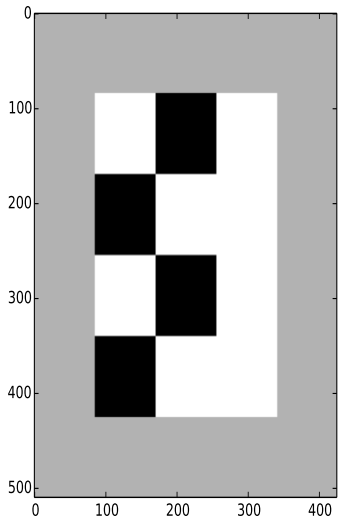
1. Représentation graphique des labyrinthes

Les labyrinthes que nous étudierons aujourd'hui prendront place sur une grille. Chaque case de la grille pourra représenter soit un morceau de mur, soit une partie de chemin.

On peut représenter le labyrinthe dans une matrice *laby* (une liste de listes d'entiers représentée à l'aide du module numpy) telle que $laby[i, j] = 0$ si la case (i, j) est un mur et 1 si cette case est un chemin.

La première chose à faire est de numéroter les colonnes du labyrinthe (de gauche à droite et de 1 à n) ainsi que les lignes (de haut en bas et de 1 à p). On considérera les colonnes 0 et $n + 1$ ainsi que les lignes 0 et $p + 1$ comme des murs : on ne les représentera pas graphiquement, mais elles nous permettrons par la suite de gérer toutes les cases de la même manière.

Les questions peuvent se traiter sans l'aide d'une représentation graphique. mais il est beaucoup plus agréable d'observer les résultats !
Les questions nécessitant le module **matplotlib** sont indiquées par un D
Définissez la matrice *laby* correspondant au labyrinthe suivant :



Nous vous proposons d'utiliser directement le code suivant, la fonction "bord" permet d'ajouter des bords à la matrice associée à l'image et la fonction "lab" de générer la matrice à afficher associée au labyrinthe. Vous pouvez à présent travailler sur une matrice de 0 et de 1.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

def bord(M):
    p=np.size(M[:,0])
    q=np.size(M[0,:])
    N=np.zeros((p+2,q+2))
    c=0.3
```

```
    for i in range(p+2):
        N[i,0]=c
        N[i,q+1]=c
    for j in range(q+2):
        N[0,j]=c
        N[p+1,j]=c
    for i in range(p):
        for j in range(q):
            N[i+1,j+1]=M[i,j]
    return N


def lab(M):
    p=np.size(M[:,0])
    q=np.size(M[0,:])
    r=min(512//p,512//q)
    img=np.zeros((r*p,r*q))
    for i in range(p):
        for j in range(q):
            for o1 in range(r):
                for o2 in range(r):
                    img[i*r+o1,j*r+o2]=int(M[i,j]*255)
    return img

M=np.array([[1,0,1],[0,1,1],[1,0,1],[0,1,1]])

img=lab(bord(M))
plt.imshow(img,cmap=plt.cm.gray)
plt.show()
# ou si l'affichage ne fonctionne pas
plt.savefig(" l'arborescence vers le bureau  /image.pdf")
```

2. Trouver le chemin le plus court vers la sortie

Un problème intéressant, lorsqu'on s'intéresse à de tels labyrinthes, est de concevoir un algorithme qui permette à partir d'une position de départ (i_d, j_d), de rejoindre une arrivée (i_a, j_a). Nous ne ferons pas d'hypothèse particulière sur le placement de ces cases qui pourront se trouver à n'importe quel emplacement de la grille.

Dans la suite, on suppose que la case (i, j) est un chemin. On dit que la case (k, l) est voisine de la case (i, j) si et seulement si elles ont un côté commun, et qu'elle est également du type chemin.

L'idée générale est la suivante : On part du point de départ (i_d, j_d), on écrit la liste des cases voisines de cette case (on les appellera cases d'ordre 1), et on regarde si l'une d'elle est la sortie. Si ce n'est pas le cas, on écrit la liste des cases voisines de ces voisines d'ordre 1, mais n'ayant pas déjà été visitées (on les appellera cases d'ordre 2). On regarde a nouveau si la sortie est atteinte ... On recommence ainsi jusqu'à ce que la sortie soit atteinte ou que toutes les cases soient visitées et qu'on soit sûr que la sortie ne peut ainsi pas être atteinte.

1. Justifier très brièvement sur l'exemple proposé que l'algorithme termine.
   Pour ne pas repasser sur une case déjà visitée, on la transformera pour la suite notre programme en mur, mais on gardera en mémoire la case la précédant, afin de pouvoir le cas échéant reconstituer le chemin. On est ainsi amené à modifier notre matrice *laby* pour pouvoir enregistrer cela.

2. Fabriquer une nouvelle matrice *lab*, correspondant au labyrinthe initial, telle que $lab[i,j] = (0,0,0)$ si la case (i,j) est un mur et $(1,0,0)$ si cette case est un chemin. Dans un premier temps, seul le premier élément du triplet nous donne des indications, les autres vont être utilisés maintenant pour la recherche de chemins dans le labyrinthe. Nous pourrons créer des fonctions "transfert" permettant de passer d'une matrice initiale à une matrice de triplets et réciproquement.

```
def tr(M):
    p=np.size(M[:,0])
    q=np.size(M[0,:])
    a,b=p//3,q//3
    M1=np.zeros((a,b))
    print(a,b)
    for i in range(a):
        for j in range(b):
            M1[i,j]=M[i,j][0]
    return M1

def tr1(M):
    p=np.size(M[:,0])
    q=np.size(M[0,:])
    M1=np.array([[[0.,0.,0.] for j in range(q)] for i in range(p)
       ])
    for i in range(p):
        for j in range(q):
            M1[i,j]=[M[i,j],0,0]
    return M1
```

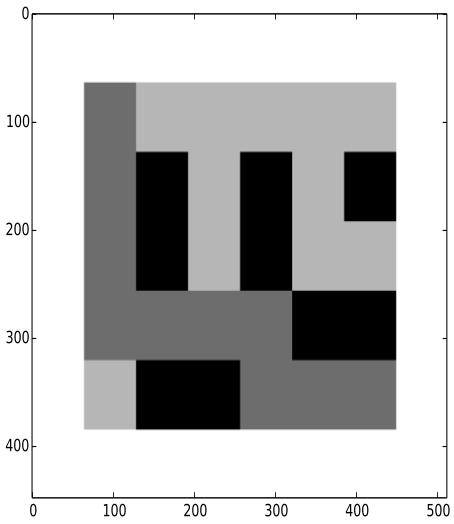
3. On suppose que la case (i,j) est la case chemin dans laquelle on se trouve à un certain moment. Ecrivez une fonction *voisin* qui crée la liste des cases voisines de cette case (i,j), et qui de plus modifie le tableau *lab* en mettant le cas échéant dans la case (k,l), si elle est voisine non visitée de (i,j), le triplet $(0,i,j)$.

4. En déduire une fonction *suivant* qui, à partir d'une liste quelconque de cases chemin, retourne la liste de leurs voisines non encore explorées.

5. Ecrire enfin une fonction *recherche* qui détermine si un couple de type (entier,entier) apparaît dans un liste d'éléments de type (entier,entier).

6. Déduire de tout cela le chemin le plus court entre la case de départ (i_d, j_d) et la case d'arrivée (i_a, j_a).

7. D Sur le labyrinthe déjà dessiné, représenter le chemin le plus court entre la case de départ (i_d, j_d) et la case d'arrivée (i_a, j_a).



3. UN BONUS : Modélisation d'un problème

Une cellule traversant une membrane poreuse a autant de difficultés à traverser qu'un individu en a pour sortir d'un labyrinthe.

Nous souhaitons déterminer une valeur approchée de la probabilité $P(p)$ de traverser cette surface, en calculant, sur un nombre conséquent d'essais, la fréquence $F(p)$ qu'il existe bien un chemin qui permette de passer du bas au haut du labyrinthe.

Générez un labyrinthe aléatoire de taille par exemple 100×100. On choisira, pour chaque case, qu'il s'agit d'un mur avec une probabilité $p \in]0, 1[$ (que vous choisirez) et d'un chemin avec une probabilité $q = (1 - p) \in]0, 1[$. On suppose également que la ligne du bas et celle du haut sont des chemins, le but étant d'essayer de traverser le labyrinthe de bas en haut.