Mini projet : **Labyrinthe élémentaire et plus court chemin.**

L'objectif de ce mini-projet est de créer un labyrinthe élémentaire (élémentaire dans le sens où on n'utilisera pas d'algorithme complexe, spécialisé dans la création de "beaux" labyrinthes), puis de déterminer (s'il existe) un plus court chemin, ainsi que sa longueur, entre deux cases choisies par l'utilisateur.

Exercice 1 : *Modélisation et représentation du labyrinthe*
Un labyrinthe sera pour nous un ensemble rectangulaire formé de cases vides (correspondant à des cases où on peut se déplacer) et de murs (cases inaccessibles). On représentera les cases vides en blanc et les murs en noir.
Tester la portion de code ci-dessous :

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

A = np.array([[2,2,2,0],[2,2,0,2],[0,2,2,2]])
plt.imshow(A,cmap="hot")
plt.colorbar()  # Ligne facultative
plt.show()
```

Essayer alors de trouver des réponses aux questions ci-dessous :
Quel est le rôle de `plt.imshow` par rapport à la matrice A? Quelle couleur représente les coefficients de A de valeur 2? de valeur 0?
Tester maintenant le code ci-dessous (à mettre à la suite du précédent) :

```
B = np.array([[1,1,2,0],[2,1,0,2],[0,1,1,2]])
plt.imshow(B,cmap="hot")
plt.colorbar()  # Ligne facultative
plt.show()
```

Quelle couleur représente les coefficients de A de valeur 1?

Les réponses aux questions ci-dessus nous permettent de comprendre comment nous allons représenter le labyrinthe :

- par une représentation matricielle où les 2 coderont les cases vides, les 0 les murs, et les 1 les cases d'un chemin parcouru,
- par une représentation graphique, immédiatement déduite de la représentation matricielle.

Exercice 2 : *Génération aléatoire d'un labyrinthe avec un nombre de murs fixé*

Principe : Supposons que notre labyrinthe soit un rectangle de taille (n, m) (où n est le nombre de lignes et m le nombre de colonnes) et que nous souhaitons p murs où $p \leqslant nm$.

Nous allons répertorier tous les couples d'indices (i, j) des coefficients de la matrice A représentant le labyrinthe et choisir successivement, aléatoirement et sans remise, p couples de coefficients (qui correspondront aux cases occupées par des murs).

1) Écrire une fonction `Tirage` prenant en entrée une liste L de q éléments, un entier naturel p, et donnant en sortie :

un message d'erreur si $p > q$,

sinon, une liste M de p éléments de L choisis successivement, aléatoirement et sans remise de la liste L.

Indication :

On pourra utiliser l'une des deux méthodes ci-dessous :

1ère méthode : *avec la méthode pop des listes*

Choisir aléatoirement les indices des éléments de L à extraire (en utilisant la fonction `randint` de la bibliothèque `random`), sachant que l'instruction

`element = L.pop(ind)` supprime l'élément d'indice `ind` de `L` et le stocke dans la variable `element`.

2ème méthode : *"à la main" (mais toujours en utilisant* `randint`*)*

On place les éléments successivement choisis à la fin de L. Pour ne pas choisir deux fois un même élément, on échange l'élément choisi avec le dernier des éléments de la liste non déjà choisis.

Illustration d'une itération (avec en gris les éléments déjà choisis) :

1	2	3	4	6	8	9	25	50	5	7	10	75

Choix aléatoire d'un élément (blanc sur fond noir)

1	2	3	4	6	8	**9**	25	50	5	7	10	75

échange avec le dernier élément non déjà choisi :

1	2	3	4	6	8	50	25	9	5	7	10	75

(le prochain élément est alors à choisir parmi 1, 2, 3, 4, 6, 8, 50 et 25 qui forment une sous-liste de L)

Remarques

Il est préférable de savoir maîtriser les **deux** méthodes.

2) Écrire une fonction Python `Liste_couples` prenant en entrée deux entiers naturels non nuls n, m et donnant en sortie la liste L des nm couples (i, j) où $0 \leqslant i \leqslant n-1$ et $0 \leqslant j \leqslant m-1$.

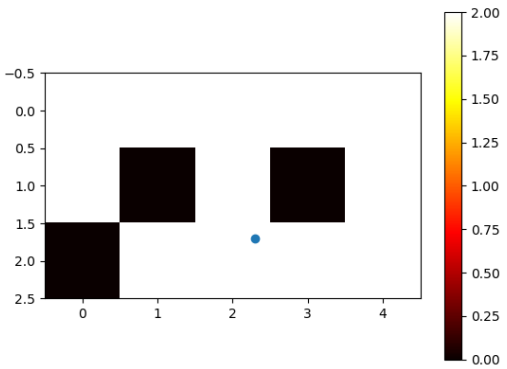
3) Déduire des deux questions précédentes une fonction Python `Genere_Laby` prenant en entrée deux entiers naturels non nuls n, m, un entier naturel p et donnant en sortie :

- un message d'erreur si $p > nm$
- sinon, une matrice A de taille (n, m) dont tous les coefficients sont égaux à 2 sauf p d'entre eux (choisis aléatoirement) qui valent 0.

Cette fonction `Genere_Laby` pourra évidemment faire appel aux deux fonctions précédentes `Tirage` et `Liste_couples`.
4) Modifier la fonction précédente (qu'on appellera encore `Genere_Laby`) de sorte qu'elle affiche une représentation graphique de A (*cf* exercice 1) et

- qu'elle stocke dans une variable `image` le résultat de `plt.imshow(...)`,
- qu'elle donne en sortie, en plus de A, la variable `image` précédemment définie.

Exercice 3 : *Lien entre coordonnées d'un point et couple d'indices de la case contenant ce point*
On souhaite que l'utilisateur puisse choisir une case du labyrinthe par un clic de souris. Supposons que A soit la matrice représentant le labyrinthe. Nous aurons donc besoin de récupérer le couple d'indices (i, j) du coefficient de A où "se trouve" le point de coordonnées (x, y) choisi par l'utilisateur. Il est donc nécessaire d'établir les relations existant entre i, j, x et y.
Prenons par exemple le cas où A est de taille $(3, 5)$ et que l'utilisateur choisisse le point de coordonnées $(2.3, 1.7)$ indiqué dans le dessin ci-dessous



alors le couple d'indice (i, j) du coefficient de A correspondant à la case contenant le point est donné par $(i, j) = (2, 2)$.
1) Écrire l'expression de i et j en fonction de x et y (les formules pourront faire intervenir la fonction `round` qui arrondit un nombre réel à l'entier le plus proche). On supposera que le point de coordonnées (x, y) n'est pas situé sur le bord d'une case.
2) En déduire une fonction Python `Indice_case` prenant en entrée un entier $n \geqslant 1$, deux réels x, y et donnant en sortie le couple (i, j) où i, j sont donnés par les formules de la question 1 précédente.

Exercice 4 : *Liste des voisins d'une case donnée*
Notons encore A la matrice représentant le labyrinthe. Par abus de langage, on confondra parfois les coefficients de A avec les cases du labyrinthe correspondantes.
On considère que chaque déplacement se fait d'une case vers le bas, vers le haut, vers la gauche ou vers la droite.
En un déplacement, on peut donc, à partir d'une case vide d'indices (i, j), accéder à 4 voisins au maximum (il peut y en avoir moins à cause des bords ou des murs). Ci-dessous, nous représentons en exemple le cas

où la case (i, j) a 4 voisins accessibles :

	$i, j-1$	
$i-1, j$	i, j	$i+1, j$
	$i, j+1$	

Pour accéder à chacun de ces voisins, on utilisera une matrice `Dep` à 4 lignes et 2 colonnes indiquant les directions à suivre pour aller de la position (i, j) à l'une des 4 positions possibles des cases voisines : `Dep = np.array([[0,-1],[-1,0],[1,0],[0,1]])`.
Ainsi, si `Pos` est la matrice `Pos=np.array([i,j])`, la position du k^e voisin sera donnée par exemple par `Pos+Dep[k]` (on rappelle que `Dep[k]` est la k-ième ligne de `Dep`). Par exemple, pour k=1, `Pos+Dep[1]` représentera la position `[i,j]+[-1,0]=[i-1,j]`. On peut aussi itérer directement sur les lignes de `Dep` si la valeur de k nous est inutile.

Écrire une fonction Python `voisins_accessibles` prenant en entrée deux indices i, j d'une case de A (correspondant à une case vide) et donnant en sortie la liste des couples d'indices des cases voisines accessibles à partir de (i, j) en un déplacement : attention aux cases situées sur les bords ou aux cases voisines inaccessibles (les murs) ! Cette fonction utilisera le tableau `Dep` mais elle ne prendra ni `Dep`, ni A en entrée, car ces matrices auront déjà été créées (et déclarée globales) au moment de l'appel de cette fonction (question 2-a de l'exercice 5).
Le pseudo-code à traduire sera donc le suivant :

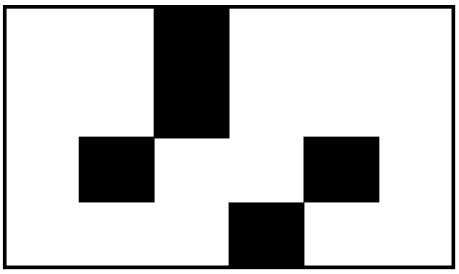
Liste_voisins_accessibles = []
Pour chaque voisin possible de (i, j)
  Si ce voisin ne "sort" pas de A et si ce voisin n'est pas un mur
    Mettre ce voisin dans Liste_voisins_accessibles

Exercice 5 : *Existence et détermination d'un plus court chemin entre deux cases : parcours en largeur.*
Passons à présent à l'algorithme permettant de déterminer (s'il existe) un plus court chemin entre deux cases vides du labyrinthe.
Le principe sera le suivant : marquer successivement des cases du labyrinthe de leur distance à la case de départ, en partant de la case de départ. Pour faire cela on effectuera ce qu'on appelle un parcours en largeur du labyrinthe (en anglais : "Breadth First Search") : on marque la première case (celle de départ), puis ses voisines accessibles (celles à distance 1), puis les voisines accessibles de ces dernières (celles à distance 2), etc.
La structure de donnée importante à utiliser est ce qu'on appelle une file, c'est-à-dire une liste dans laquelle on fera plusieurs fois les opérations suivantes :

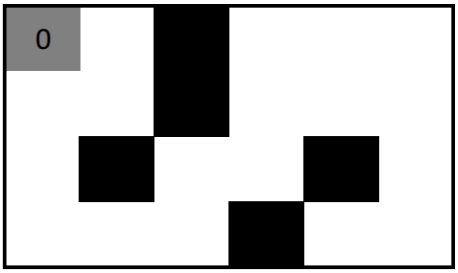
- ajouter un élément en fin de liste,
- traiter puis supprimer l'élément en début de liste.

Cette file contiendra successivement le numéro de la case correspondant au départ (à distance 0 de la case de départ), les numéros des cases à distance 1 du départ, les numéros des cases à distance 2, etc.

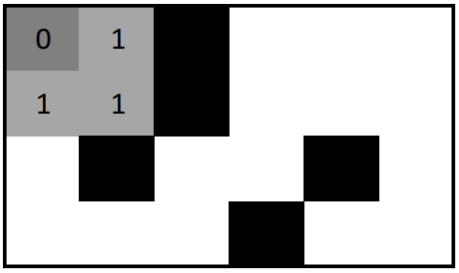
On a illustré le fonctionnement du parcours en largeur sur un exemple (voir ci-dessous) où la case de départ est l'entrée du labyrinthe (case du coin supérieur gauche) et la case d'arrivée est la case du coin inférieur droit.
Sur le labyrinthe, on a marqué les cases traitées de leur distance à l'entrée du labyrinthe. Enfin, les cases en gris foncée sont les cases en cours de traitement (celles dont on cherche les cases voisines à insérer à l'étape suivante dans la file).



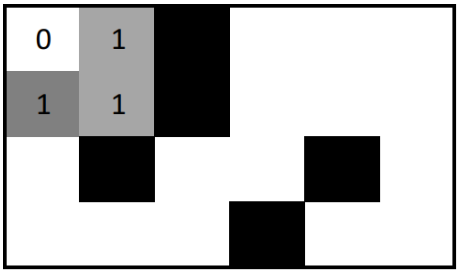
File = []



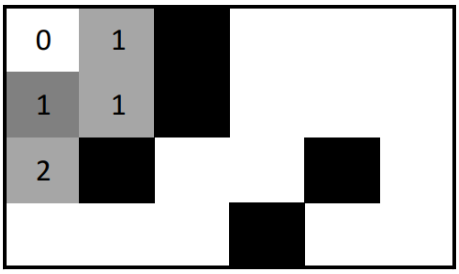
Ajout de $(0,0)$
File = $[(0,0)]$



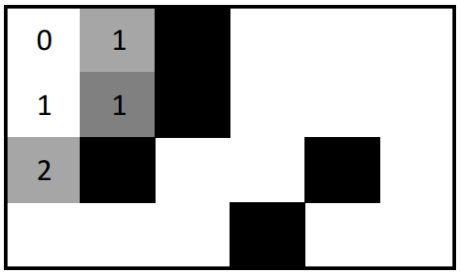
Ajout des voisins de $(0,0)$
File = $[(0,0),(1,0),(1,1),(0,1)]$



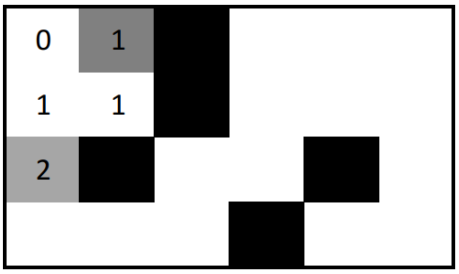
Suppression de $(0,0)$
File = $[(1,0),(1,1),(0,1)]$



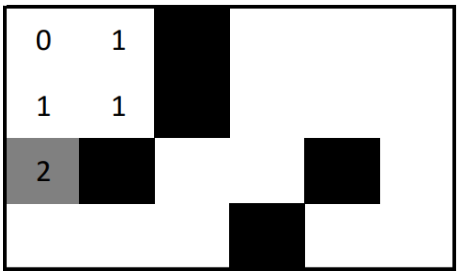
Ajout du voisin de $(1,0)$
File = $[(1,0),(1,1),(0,1),(2,0)]$



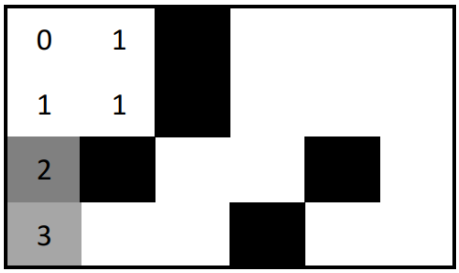
Suppression de $(1,0)$
File = $[(1,1),(0,1),(2,0)]$



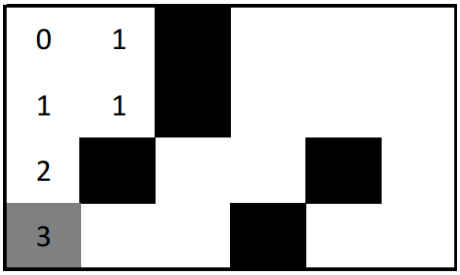
Suppression de $(1,1)$
File $= [(0,1),(2,0)]$



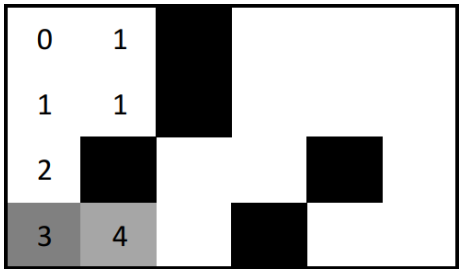
Suppression de $(0,1)$
File $= [(2,0)]$



Ajout du voisin de $(2,0)$
File $= [(2,0),(3,0)]$



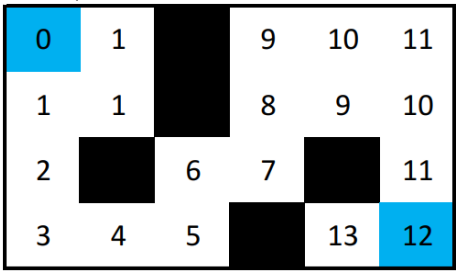
Suppression de $(2,0)$
File $= [(3,0)]$



Ajout du voisin de $(3,0)$
File $= [(3,0),(3,1)]$

etc.

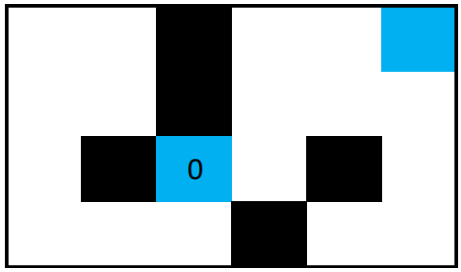
Si l'on poursuit le marquage complet de toutes les cases accessibles du labyrinthe, on aboutit finalement au marquage suivant (avec la case de départ et d'arrivée en bleu) :

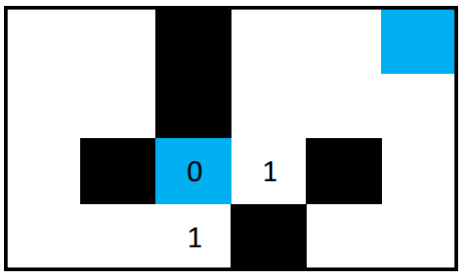


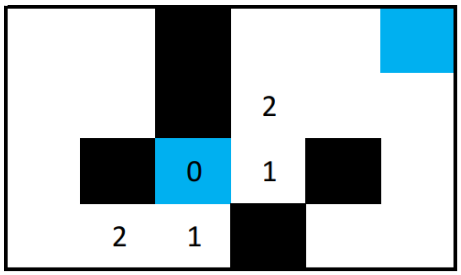
File $= []$ (la file est vide à la fin de l'algorithme dans ce cas)

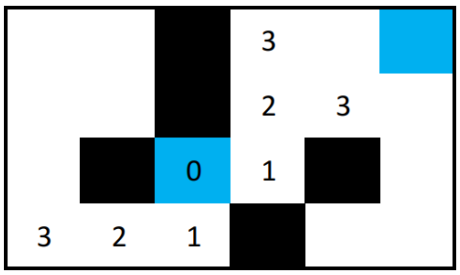
Pour parcourir les voisins d'une case donnée, on utilisera la fonction de l'exercice 4.

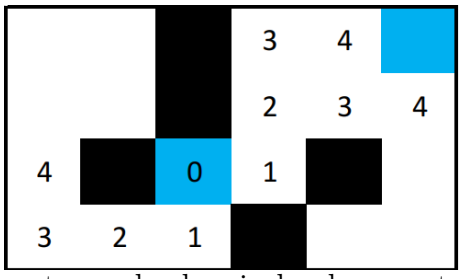
Ci-dessous, un autre exemple avec deux autres cases pour le départ et l'arrivée (on arrête les marquages dès que l'on a marqué la case d'arrivée) :

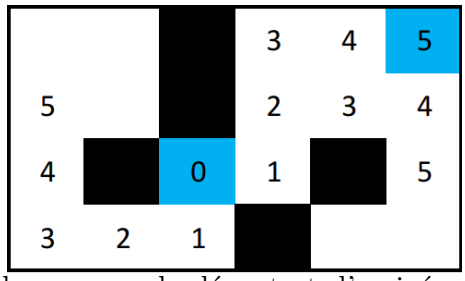




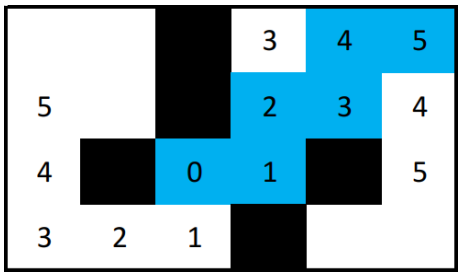




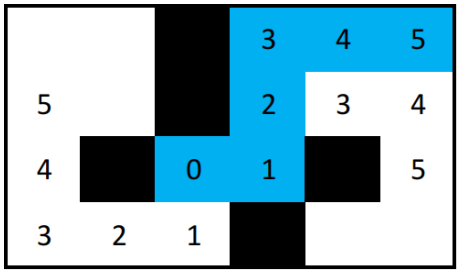




Pour tracer le chemin le plus court reliant les deux cases de départ et d'arrivée, il suffira de partir de la case d'arrivée puis de "remonter" jusqu'à l'*unique* case marquée 0 (c'est-à-dire la case de départ) en suivant les distances décroissantes (5, 4, 3, 2, 1 et 0 dans notre cas). Ici, il y a deux chemins possibles (indiqués en bleu)

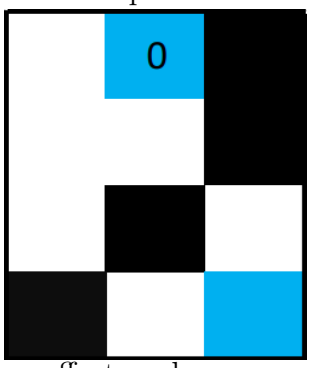


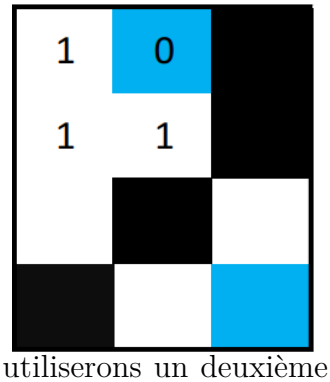
ou

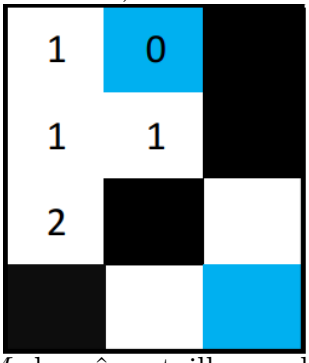


On peut aussi être dans la situation où la case d'arrivée est inaccessible à partir de la case de départ. Dans

ce cas, on marque toutes les cases accessibles possibles, et l'algorithme se termine lorsque la file est vide (voir l'exemple ci-dessous aves les cases de départ et d'arrivée toujours en bleu).







Pour effectuer les marquages, nous utiliserons un deuxième tableau M de même taille que la matrice A représentant le labyrinthe. Les cases non marquées de ce tableau seront codées par la valeur -1.

1) *Initialisation du tableau de marquage*

Écrire une fonction Python `Initialise_M` prenant en entrée deux entiers naturels n, m non nuls (correspondant respectivement au nombre de lignes et au nombre de colonnes de A) et donnant en sortie une matrice M de taille (n, m) dont tous les coefficients valent -1.

2) *Algorithme de parcours en largeur et plus court chemin*

a) Écrire une fonction Python `Parcours_largeur` qui :

- prend en entrée deux couples d'indice (i, j) et (p, q) de cases de A,
- initialise le tableau de marquage M avec de plus $M[i, j] = 0$ (marquage de la case (i, j) de départ),
- marque dans M toutes les cases partant de (i, j) de leur distance à la case (i, j) jusqu'à marquer la case (p, q) (*cf* deuxième exemple ci-dessus), en suivant l'algorithme de parcours en largeur décrit ci-dessus.

  On utilisera une file comme il a été expliqué plus haut. Cette file sera une liste initialisée à $[(i, j)]$ dans laquelle on va insérer toutes les cases en cours (ou en attente) de traitement et retirer celles qui ont déjà été traitées (*cf* le premier exemple illustré).

  Le pseudo-code à traduire sera alors essentiellement le suivant :

  > Tant que la case d'indices (p, q) de M n'a pas été marquée et que la file n'est pas vide
  > prendre et supprimer la case (a, b) au début de la file
  > `distance =` $M[a, b]$ `+ 1`
  > mettre en fin de file le ou les voisins accessibles de cette case non marqué(s) ...
  > ...et les marquer de la valeur `distance` dans M

- La fonction donnera le tableau de marquage M obtenu en sortie.

Important : Cette fonction créera la matrice `Dep` définie à l'exercice 4, et déclarera cette variable comme une variable globale (car elle devra être accessible à la fonction `voisins_accessibles` qu'elle appellera).

b) Écrire une fonction Python `chemin` qui affiche la liste des cases d'un chemin le plus court de la case (p, q) à la case (i, j). Plus précisément, cette fonction devra :

- prendre en entrée :
  - deux couples (i, j) et (p, q) de cases d'un labyrinthe représenté par une matrice de codage A,
  - le tableau M décrit dans la question précédente (et donné en sortie par la fonction précédente avec (i, j) et (p, q) en entrée).

- donner en sortie un message d'erreur si la case (i, j) est inaccessible par un chemin du labyrinthe partant de la case (p, q),
- sinon, donner la liste L des couples d'indices des cases d'un chemin le plus court de (p, q) à (i, j) (avec (p, q) et (i, j) inclus dans cette liste).

La liste L des couples sera construite à partir du tableau M de marquages en partant de la case (p, q) et en suivant successivement des cases voisines à distances strictement décroissantes de la case (i, j), jusqu'à la case (i, j) (l'unique case marquée 0).

Par exemple, si M est donnée par $\begin{pmatrix} -1 & -1 & -1 & 3 & 4 & 5 \\ 5 & -1 & -1 & 2 & 3 & -1 \\ 4 & -1 & 0 & 1 & -1 & -1 \\ 3 & 2 & 1 & -1 & -1 & -1 \end{pmatrix}$ avec $(i, j) = (2, 2)$ et $(p, q) = (0, 5)$, un plus court chemin serait (indiqué en gris) :

$$\begin{pmatrix} -1 & -1 & -1 & 3 & 4 & 5 \\ 5 & -1 & -1 & 2 & 3 & -1 \\ 4 & -1 & 0 & 1 & -1 & -1 \\ 3 & 2 & 1 & -1 & -1 & -1 \end{pmatrix}$$

ou :

$$\begin{pmatrix} -1 & -1 & -1 & 3 & 4 & 5 \\ 5 & -1 & -1 & 2 & 3 & -1 \\ 4 & -1 & 0 & 1 & -1 & -1 \\ 3 & 2 & 1 & -1 & -1 & -1 \end{pmatrix}$$

et on aurait, dans le premier cas, $L = [(0, 5), (0, 4), (0, 3), (1, 3), (2, 3), (2, 2)]$ (et dans le deuxième : $L = [(0, 5), (0, 4), (1, 4), (1, 3), (2, 3), (2, 2)]$).

Exercice 6 :

1) a) Écrire une fonction Python dont le rôle va être de coder les cases d'un chemin indiqué en entrée.
Cette fonction prendra en entrée une liste L de couples d'indices de cases (libres) d'un chemin dans le labyrinthe (telle que donnée en sortie par la fonction de la question 2-b de l'exercice 5). Elle déclarera globale la variable A (où A est la matrice qui code le labyrinthe).
Elle modifiera les coefficients de A dont les indices sont indiqués dans L (pour les conventions de codage, revoir l'exercice 1) b) Écrire une fonction effectuant la procédure inverse : à partir de L, modifier les coefficients de A indiqués par L de sorte qu'ils correspondent à nouveau à des cases blanches (c'est-à-dire sans mur et non parcourue par un chemin).
2) a) Écrire la fonction ci-dessous dans votre éditeur, à la suite des précédentes fonctions écrites jusqu'ici :

```
def Trace(image):
    global compteur, coord, A
    compteur = 0
    coord = []
    n = A.shape[0]
    def clic(event):
        global compteur, coord, A
        x, y = event.xdata, event.ydata
        i, j = Indice_Case(float(x), float(y))
        if A[i,j] == 0:
            print("Cette case est un mur")
            return
        compteur += 1
        if compteur%3 > 0:
            coord.append((i,j))
            if compteur%3 == 2:
                for couple in coord:
                    A[couple] = 1
                image.set_data(A)
                plt.draw()
        else:
            for couple in coord:
                A[couple] = 2
            image.set_data(A)
            plt.draw()
            coord = []
    plt.connect('button_press_event', clic)
```

puis faire appel à la fonction `Genere_Laby` (question 4 de l'exercice 2) et à la fonction `Trace` précédente en tapant dans la console

```
A, img = Genere_Laby(10,15,50)
Trace(img)
```

Cliquer alors plusieurs fois sur des cases blanches du labyrinthe et constater ce qui se passe.
Quelques explications sur la fonction `Trace` :

- La dernière ligne `plt.connect('button_press_event', clic)` signifie qu'on va associer un évènement à chaque appui d'un bouton de la souris et que cet évènement est défini par la fonction `clic` définie un peu plus haut dans le code. Autrement dit, chaque fois qu'on cliquera sur la souris à un endroit du graphique, la fonction `clic` sera exécutée.

- Les variables `event.xdata` et `event.ydata` font référence aux coordonnées x et y du point du graphique sur lequel on a cliqué avec la souris.
- `image.set_data(A)` signifie qu'on met à jour l'instruction `plt.imshow(...)` que contient `image` avec `A` comme nouvel argument.
- L'instruction `plt.draw()` a ici pour effet de mettre à jour le graphique.

Essayer de comprendre tout le code de la fonction `Trace`. En particulier, quel est le rôle de la variable `compteur` ?

b) Une fois la fonction `Trace` bien comprise, essayer de la modifier pour écrire une fonction `Trace_Chemin` prenant toujours en entrée `image`. Si cette fonction est appelée à la suite de `LabyAleatoire`, l'utilisateur doit pouvoir voir s'afficher un plus court chemin d'une case du labyrinthe à une autre, cases qu'il aura choisies avec la souris (si un tel chemin n'existe pas, il doit voir s'afficher un message d'erreur). Il lui sera aussi indiqué à chaque fois la longueur d'un tel chemin et il pourra répéter l'expérience autant de fois qu'il le veut.

Cette fonction `Trace_Chemin` fera appel aux fonctions des questions 2-a et 2-b de l'exercice 5, ainsi qu'aux fonctions des questions 1) a) et 1) b) de cet exercice 6. Elle pourra avoir pour squelette :

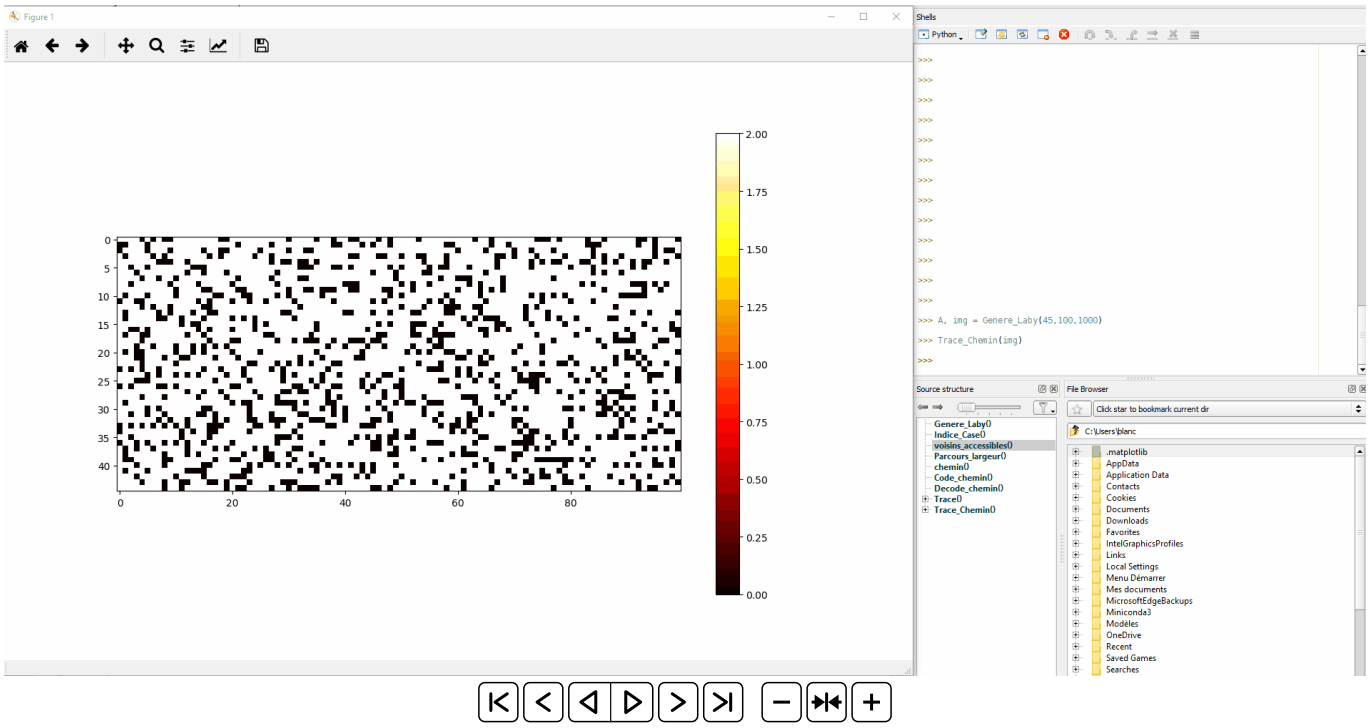
```
def Trace_Chemin(image):
    global compteur, ... ...
    .
    .
    .
    def clic(event):
        global compteur, ... ...
        .
        .
        .
        if compteur%3 > 0:
            .
            .
            if compteur%3 == 2:
                .
                .
                .
        else:
            .
            .
            .
    plt.connect('button_press_event', clic)
```

Une fois la fonction `Trace_Chemin` écrite et fonctionnant correctement, vous pourrez supprimer la fonction `Trace` de votre programme.

Exercice 7 : *Synthèse*

Écrire à présent la fonction Python principale du projet (celle articulant toutes les précédentes) :
cette fonction ne prendra pas d'argument en entrée, demandera à l'utilisateur la taille (n, m) du labyrinthe ainsi que le nombre de murs p. Elle affichera le labyrinthe puis permettra à l'utilisateur de cliquer plusieurs fois sur deux cases pour avoir le plus court chemin entre ces deux cases, ainsi que sa longueur.
Cette fonction fera donc appel à la fonction `Genere_Laby` de l'exercice 2 (question 4) et à la fonction `Trace_Chemin` de l'exercice 6.
Proposer également à l'utilisateur de définir plutôt le pourcentage de murs sur le nombre total de cases (au lieu du nombre de murs) et adapter la valeur de p en fonction de la réponse fournie par l'utilisateur.

Voici ci-dessous à quoi pourra ressembler l'exécution du projet :



Extension possible :

Supposons que l'entrée du labyrinthe soit la case tout en haut à gauche du labyrinthe et la sortie la case tout en bas à droite du labyrinthe.
Avec n, m fixés (largeur et longueur du labyrinthe), déterminer pour diverses valeurs de p la moyenne des longueurs d'un plus court chemin de l'entrée à la sortie sur plusieurs labyrinthes aléatoires associés aux paramètres m, n et p (on ignorera les labyrinthes où un tel chemin n'existe pas ou ceux où l'entrée ou la sortie sont obstruées par un mur).
Représenter graphiquement ces moyennes sur un graphique en fonction de p (le paramètre variant). Une nouvelle fonction Python pourra être éventuellement réalisée pour automatiser tout cela. Commenter les résultats obtenus.