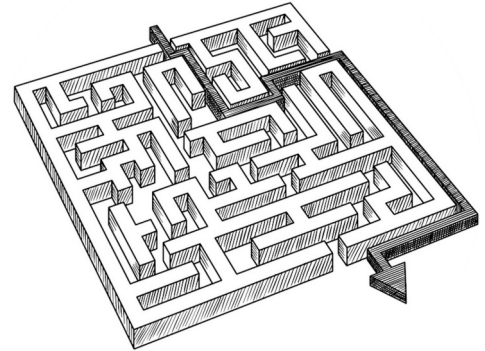


Le projet

Dans ce projet, on cherche à concevoir puis implanter un algorithme permettant à un robot de sortir de façon autonome d'un labyrinthe quelconque.

Voici les différentes étapes que vous allez réaliser :

- **Etape 1** : Représentation d'un labyrinthe
- **Etape 2** : Création d'une classe **Labyrinthe**
- **Etape 3** : Création d'une fonction pour rechercher les déplacements possibles à partir d'une case donnée.
- **Etape 4** : Recherche du chemin de façon autonome
- **Etape 5 (facultative)** : Construction d'une visualisation graphique



A la fin de chaque étape, appeler fièrement votre enseignante pour lui montrer votre travail!

Etape n° 1

On modélise un labyrinthe par un tableau à deux dimensions à n lignes et m colonnes avec n et m des entiers strictement positifs.

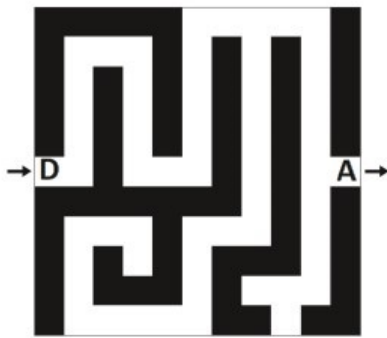
Les lignes sont numérotées de 0 à $n - 1$ et les colonnes de 0 à $m - 1$.

La case en haut à gauche est repérée par $(0, 0)$ et la case en bas à droite par $(n - 1, m - 1)$.

Dans ce tableau :

- 0 représente une case vide, hors case de départ et arrivée,
- 1 représente un mur,
- 2 représente le départ du labyrinthe,
- 3 représente l'arrivée du labyrinthe.

Ainsi, en Python, le labyrinthe ci-dessous est représentée par le tableau de tableaux **tab1**.

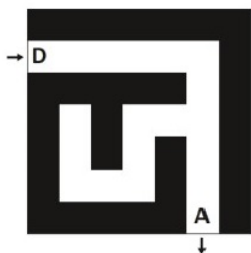


```
tab1 = [[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1],
        [1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [2, 0, 1, 0, 0, 0, 1, 0, 1, 0, 3],
        [1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1],
        [1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1]]
```

1. Le labyrinthe ci-dessous est censé être représenté par le tableau de tableaux **tab2**.

Cependant, dans ce tableau, un mur se trouve à la place du départ du labyrinthe.

Écrire une instruction permettant de placer le départ au bon endroit dans **tab2**.



```
tab2 = [[1, 1, 1, 1, 1, 1, 1],
        [1, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 1, 1, 0, 1],
        [1, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 1],
        [1, 1, 1, 1, 1, 3, 1]]
```

2. Écrire les instructions permettant d'obtenir le nombre de lignes et le nombre de colonnes du labyrinthe représenté par le tableau **tab2**.

Etape n° 2

On souhaite donc écrire une classe `Labyrinthe` comme ceci

```
class Labyrinthe:
    def __init__(self, tab):
        self.tab = tab # tableau représentant le labyrinthe

lab1 = Labyrinthe(tab1)
lab2 = Labyrinthe(tab2)
```

1. Compléter et écrire les méthodes `nb_lignes` et `nb_colonnes` de la classe `Labyrinthe` permettant d'obtenir respectivement le nombre de lignes et le nombre de colonnes du labyrinthe représenté par `tab2`. Tester vos méthodes.

```
def nb_lignes(self):
    return ....

def nb_colonnes(self):
    return .....
```

2. Écrire les méthodes `depart` et `arrivee` qui renvoient respectivement la position de la case du départ et de l'arrivée sous la forme d'un tuple. Tester vos méthodes.
3. Écrire une méthode `est_valide(self, i, j)` qui prend en paramètres deux nombres entiers `i` et `j` et qui renvoie `True` si le couple `(i, j)` correspond à des coordonnées valides pour un labyrinthe et `False` sinon.

On donne ci-dessous des exemples d'appels.

```
>>> lab1.est_valide(5, 2)
True
>>> lab1.est_valide(-3, 12)
False
```

4. Écrire une méthode `nb_cases_vides(self)` qui renvoie le nombre de cases vides d'un labyrinthe (comprenant donc l'arrivée et le départ).
Par exemple, l'instruction `lab1.nb_cases_vides()` doit renvoyer la valeur 58.

Etape n° 3

On va devoir parcourir les cases vides de proche en proche. Lors d'un tel parcours, afin d'éviter de tourner en rond, on choisit de marquer les cases visitées.

Pour cela, on remplace la valeur d'une case visitée dans le tableau représentant le labyrinthe par la valeur 4.

1. Écrire la méthode `est_visite(self, i, j)` qui modifie l'attribut `tab` du labyrinthe en notant la case `(i, j)` comme visitée en lui attribuant le chiffre 4. Tester votre fonction.
2. On dit que deux cases d'un labyrinthe sont voisines si elles ont un côté commun. On considère une méthode `liste_voisines_libres(self, i, j)` qui prend en arguments deux entiers `i` et `j` représentant les coordonnées d'une case du labyrinthe.

Cette fonction renvoie la liste des coordonnées des cases voisines de la case de coordonnées `(i, j)` qui sont valides, non visitées et qui ne sont pas des murs. L'ordre des éléments de cette liste n'importe pas. On donne ci-dessous des exemples d'appels.

```
>>> lab1.liste_voisines(1, 2)
[(1, 1), (1,3)] # peu importe l'ordre
>>> lab1.liste_voisines(5, 9)
[(4, 9), (6,9), (5,10)]# peu importe l'ordre
>>> lab1.liste_voisines(5, 0)
[(5, 1)]
```

Écrire cette méthode.

Etape n° 4

Dans la suite, on appellera solution du labyrinthe : un chemin allant du départ à l'arrivée sans repasser par la même case.

Pour déterminer la solution d'un labyrinthe, on parcourt les cases vides de proche en proche.

On marquera la valeur d'une case visitée dans le tableau représentant le labyrinthe par la valeur 4. On rappelle aussi, qu'une case est dite libre si elle a pour valeur 1, 2 ou 3.

L'objectif est d'écrire un programme qui détermine s'il existe un chemin de l'entrée vers la sortie en se déplaçant vers le haut, le bas, la gauche ou la droite (mais pas en diagonale).

L'idée est de parcourir le labyrinthe depuis l'entrée, en utilisant une pile pour stocker le chemin parcouru au fur et à mesure, pour pouvoir dépiler lorsque le chemin n'aboutit pas et redémarrer sur une autre voie.

1. Un exemple sera sans doute plus efficace qu'un long discours, pour cela aller consulter le document disponible sur le moodle nommé **exemple**.
2. Si, au cours de l'exécution la pile se trouve vide, que cela signifie-t-il ?
3. Écrire et importer un module comportant une classe **Pile** avec les méthodes dont vous aurez besoin.
4. Écrire une fonction, une méthode ou un programme permettant de trouver la solution d'un labyrinthe.

Etape n° 5

1. Aller consulter le site suivant présentant le module **pygame** :

<http://sdz.tdct.org/sdz/interface-graphique-pygame-pour-python.html>

2. Construire une représentation « graphique » d'un labyrinthe selon vos envies et représenter ensuite une résolution visuelle d'une solution du labyrinthe.

Par exemple :

