LABYRINTHES

L'objectif de ce TD est de réaliser automatiquement un labyrinthe (avec une seule solution), puis de le résoudre.

1 PRÉAMBULES

Chargement de bibliothèques :

```
import numpy as np                  # grille du labyrinthe
from random import *                # c'est mieux pour un labyrinthe\ldots
from matplotlib.pyplot import *     # pour afficher le labyrinthe
ion()                               # pour un tracé interactif
```

Modélisation : Le labyrinthe sera modélisé par une grille de taille $n \times p$: la case contient 1 si c'est un chemin et 0 sinon.

Variables globales : Pour simplifier et éviter trop d'arguments aux fonctions, nous utiliserons certains variables globales

- `laby` désigne le tableau numpy qui représente le labyrinthe,
- `width` et `height` désignent la largeur et la hauteur du labyrinthe (nombre de cases),
- `depart` désigne les coordonnées du point de départ, et `arrivee` désigne les coordonnées du point d'arrivée,
- `position` désigne la position courante (couple abscisse, ordonnée),
- `dir` désigne la direction actuelle du chemin suivant (4 directions possibles suivant le code suivant)

Une autre variable globale sera introduite ultérieurement.

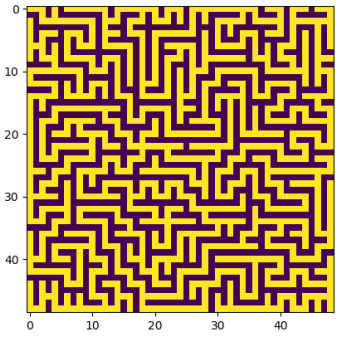
⚠ on n'oubliera pas le mot clef **global** dans la fonction, si on souhaite modifier une variable globale.

Codage des directions :

- 'G' : reculer suivant l'axe des abscisses,
- 'D' : avancer suivant l'axe des abscisses,
- 'H' : reculer suivant l'axe des ordonnées,
- 'B' : avancer suivant l'axe des ordonnées.

2 ON CREUSE...

L'idée de base est de creuser en avançant de deux cases à la fois (pour laisser au moins une largeur de terre entre deux couloir).
À chaque fois que l'on avance, on a le choix d'avancer dans trois directions différentes (à gauche, à droite ou tout droit). On stocke ces trois possibilités dans une liste globale `amorces` qui contient donc tous les points de départ des chemins futurs.
Puis pour continuer à creuser, on choisit dans la liste une des amorces quelconque (que l'on vient de créer ou une plus ancienne, suivant une stratégie de votre choix) et on creuse dans la direction indiquée.
Bien sûr, pour éviter de tourner en rond, on interdira aux chemins de se croiser : notre galerie ne doit jamais aboutir dans un chemin déjà existant. De même, on s'interdira de sortir de la grille.



La liste `amorces` contient donc des couples `(position,dir)`, tels que `position` désigne les coordonnées du points à partir duquel creuser, et `dir` donne le code de la direction dans laquelle creuser. Quand la liste est vide, c'est que le labyrinthe est terminé : il n'y a plus d'endroit où creuser.

Pour vous aider (mais vous pouvez vous en passer), on détaille ci-après un découpage du problème en plusieurs fonctions.

1) Créer une fonction **valide**(`pos`) qui teste si la position `pos` est dans le labyrinthe (renvoie **True** ou **False**).

2) Créer une fonction **suivant**(`pos,dir,nb`) qui renvoie la position correspondant à l'avancement de `nb` cases dans la direction `dir` depuis `pos`.

La fonction ne teste pas si la position renvoyée est valide.

3) Créer une fonction **creuseSuivant**() qui creuse de deux cases...
La fonction utilise les variables globales `position` et `dir` qui indiquent respectivement la position de la taupe et la direction dans laquelle creuser.
Avant de creuser, on vérifiera que cela ne fait pas sortir du labyrinthe et n'aboutit pas dans une galerie déjà existante (sinon, on ne fait rien).
Après avoir creusé les deux cases, on ajoutera quatre éléments à la liste `amorces` correspondant aux quatre directions possibles pour continuer à creuser[1]. La fonction s'arrête ici.

4) Créer une fonction **creuser**(`depart`) qui crée un labyrinthe rempli de 0, et qui initialise le parcours à la position `depart`.
On place dans la liste `amorces`, les 4 directions possibles pour partir, puis, tant qu'il reste des amorces, on en choisit une (suivant un protocole au choix) et on creuse de deux cases dans la direction choisie, avant de prendre une autre amorce.

On pourra afficher le labyrinthe à la fin.

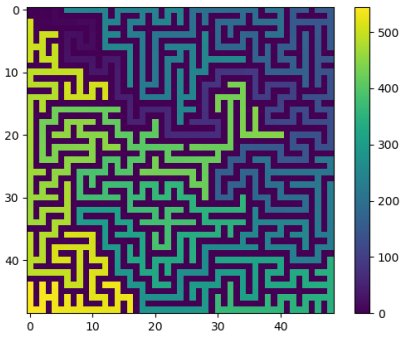
Quelques commandes

```
laby = np.zeros((width,height))        # initialise le labyrinthe
imshow(laby)                           # affiche le labyrinthe
```

La façon avec laquelle vous choisirez vos amorces aura une grande influence sur l'allure de votre labyrinthe. Vous pourrez le vérifier par vous-même.

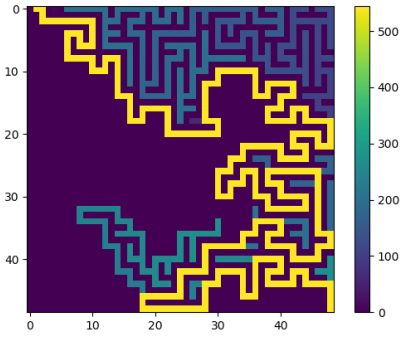
Amélioration : imposer que l'arrivée soit le bout d'une chemin (que le chemin de ne prolonge pas au delà).

3 Profondeur



Le but de cette partie est de colorier le chemin en fonction de sa distance par rapport au point de départ.
Plus simplement, au lieu de mettre le numéro 2 dans les cases du chemin, nous mettrons leur distance par rapport à l'origine (+2).
Ainsi la case `depart` contient la valeur 2, les cases adjacentes (creusées) contiennent 3... et ainsi de suite.
Comme nous avons pris soin d'éviter les recoupements de galerie, il n'y aura qu'une seule distance possible pour chaque case.
Pour réaliser cela, on part du labyrinthe déjà creusé et on suit les galeries.
On pourra utiliser la même méthode que pour creuser (avec le système d'amorces), mais on n'ira que dans les directions déjà creusées.
On pourra réaliser un *parcours en largeur* qui consiste à chaque fois, à repartir de l'amorce la plus ancienne.
La commande `colorbar()` donne l'échelle des couleurs sur l'image.
On remarquera que la façon de choisir les amorces peut changer radicalement la répartition des profondeurs. Vous pourrez réfléchir à celle qui est la plus adaptée à un labyrinthe.

4 Solution



À présent, on cherche simplement à mettre en valeur le chemin solution.
Pour cela, on colorie comme précédemment le labyrinthe jusqu'à atteindre l'arrivée (on peut s'arrêter dès que l'on a trouvé l'arrivée, car tous les chemins non explorés sont alors des impasses).
Puis, à partir de l'arrivée, on revient en arrière en suivant le chemin des distances strictement décroissantes.

On pourra prolonger le plaisir en visitant le blog de Mike Bostock https://bost.ocks.org/mike/algorithms/ dont je me suis inspiré pour réaliser ce TD.
Cela peut constituer une belle base pour un projet d'algorithmique en 2ème année.

[1] La direction qui revient en arrière sera invalidée immédiatement quand on la testera car elle aboutit à une galerie existante. Il n'est donc pas utile de se fatiguer à la retirer.