Activité 07 - Labyrinthes

Equipe Pédagogique LU1IN0*1

Consignes : Cette activité se compose d'une première partie guidée, suivie de suggestions. Il est conseillé de traiter en entier la partie guidée avant de choisir une ou plusieurs suggestions à explorer.

L'objectif de cette activité est l'étude, la résolution et la génération de labyrinthes, en utilisant, quand c'est pertinent, des fonctions récursives.

1 Partie guidée : Chemins et Labyrinthes

On suppose qu'on travaille sur une **grille** à coordonnées entières. L'origine de la grille a pour coordonnées $(0, 0)$, la case directement au Nord de l'origine a pour coordonnées $(0, 1)$, celle directement à l'Est de l'origine a pour coordonnées $(1, 0)$.

Un "déplacement" depuis une case se fait dans une des quatres directions cardinales (Nord, Est, Sud, Ouest) et aboutit sur la case adjacente à la case initiale, dans la direction choisie.

Ainsi si on se déplace vers le Sud depuis la case de coordonnées $(3, 3)$, on arrive dans la case de coordonnées $(3, 2)$, et si, de cette case on se déplace vers l'est, on arrive aux coordonnées $(4, 2)$.

On définit les alias de types suivants :

```
Coord = Tuple[int, int]
Dir = str
# un element de Dir est soit "N", soit "S", soit "E", soit "O"
```

Le type `Coord` des *coordonnées* décrit les couples (abscisses, ordonnées) des cases d'une grille. Le type `Dir` des *directions* décrit la collection des caractères `"N"`, `"E"`, `"S"`, `"O"`.

Question 1. Donner une fonction `deplace` qui prend en entrée des coordonnées `c` et une direction `d` et qui renvoie les coordonnées de la case dans laquelle on se trouve après un déplacement dans la direction `d` depuis la case de coordonnées `c`.

```
ori : Coord = (0, 0)
p1 : Coord = (3, 3)
p2 : Coord = (0, 2)

assert deplace(ori, "N") == (0, 1)
assert deplace(p1, "E") == (4, 3)
assert deplace(deplace(p2, "N"), "S") == p2
```

Chemins. Pour représenter *un chemin* (c'est-à-dire, un itinéraire dans la grille), on utilise une **liste de directions**.

Par exemple `["N", "N", "E", "S"]` représente le chemin qui consiste à aller deux fois de suite vers le nord, puis une fois vers l'est, puis une fois vers le sud.

Question 2. Donner une fonction **récursive** `deplace_chemin` qui prend en entrée des coordonnées `c` est un chemin `ch` et qui renvoie les coordonnées de la case dans laquelle on arrive après avoir suivi toutes les directions de `ch`, dans l'ordre.

```
assert deplace_chemin(ori, ["N", "N"]) == p2
assert deplace_chemin(ori, ["N", "E", "S", "O"]) == ori
assert deplace_chemin(ori, []) == ori
```

Cases. On veut représenter un labyrinthe composé de pièces carrées (cases) entourées d'au plus quatre murs (un dans chaque direction), quand un mur n'est pas présent dans une direction, le passage est libre vers la case voisine.

Ainsi, une *case* d'un labyrinthe est un quintuplet `(n, e, s, o, nat)` composé de quatre booléens qui indique, dans l'ordre si le passage est libre vers le Nord (`n`), l'Est (`e`), le Sud (`s`) et l'Ouest (`o`) depuis cette case, et une chaîne de caractères `nat`, indiquant si la case possède des particularités, par exemple `nat` vaut `"ENTREE"` si la case une entrée et `"SORTIE"` si c'est une sortie et `nat` vaut `""` pour une case "normale".

Ainsi `(True, True, False, False, "")` représente une case qui possède un accès vers le Nord et vers l'Est, mais qui est bloquée par un mur au Sud et à l'Ouest.

Et `(False, False, True, False, "ENTREE")` représente une case qui est une entrée du labyrinthe et qui est bloquée par des murs dans toutes les directions sauf vers le Sud.

On donne l'alias :

```
Case = Tuple[bool,bool,bool,bool,str]
```

Labyrinthes. Un *labyrinthe* `la` est une liste de listes de cases. Chaque élément `la[i]` de `la` représente une colonne du labyrinthe, en partant de la colonne d'abscisse 0 (la plus à l'Ouest). Chaque élément `la[i][j]` de `la[i]` est une case de la colonne, en partant de la case d'ordonnée 0 (la plus au Sud).

Ainsi `la[x][y]` contient la case de coordonnées (x, y) du labyrinthe.

Dans la suite, on supposera que tous les labyrinthes sont *rectangulaires*, c'est-à-dire que toutes les colonnes ont autant d'éléments. La *largeur* d'un labyrinthe est sa longueur (son nombre de colonnes), sa *hauteur* est la longueur d'une de ses colonnes.

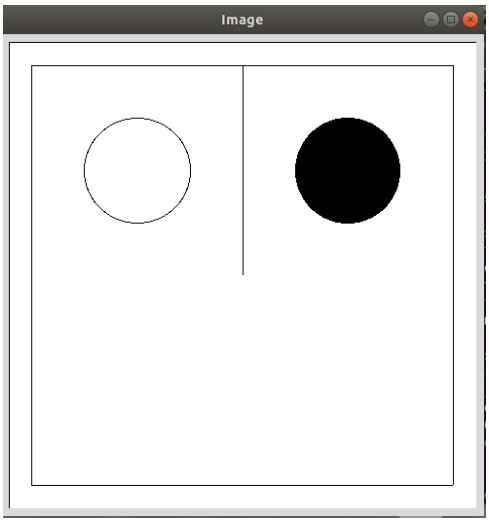
On définit l'alias suivant :

```
Laby = List[List[Case]]
```

Et trois exemples de labyrinthes[1], pour faire des essais et des tests :

1. ```
   laby1 : Laby = [[(True, True, False, False, ""),
                    (False, False, True, False, "ENTREE")],
                   [(True, False, False, True, ""),
                    (False, False, True, False, "SORTIE")]]
   ```

   Qu'on peut représenter graphiquement (cf. Section 6) (avec un disque blanc pour l'entrée et noir pour la sortie) par :

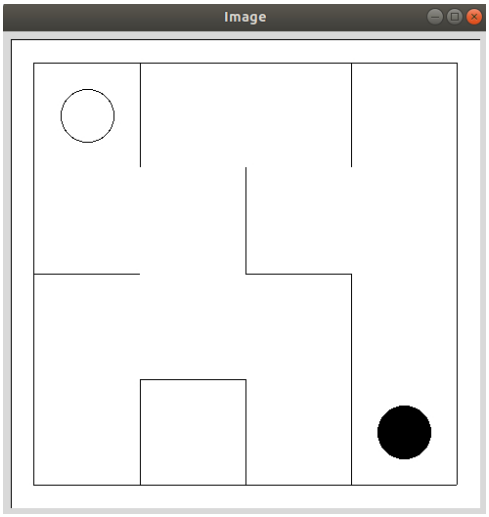


[1] disponibles dans un fichier fourni sur Moodle.
   ```

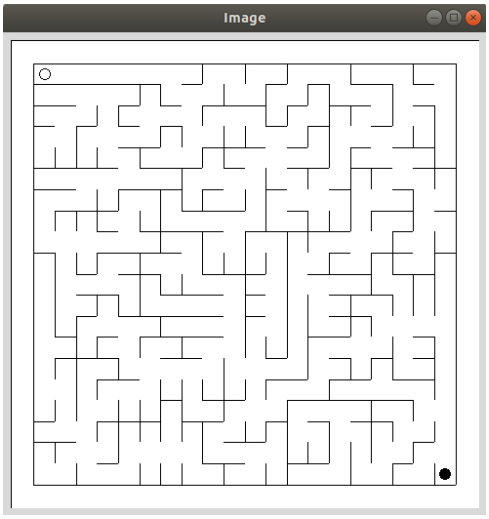
2.

```
laby2 : Laby = [
    [(True, False, False, False, ""),
     (False, True, True, False, ""),
     (True, True, False, False, ""),
     (False, False, True, False, "ENTREE")],
    [(False, False, False, False, ""),
     (True, True, False, True, ""),
     (True, False, True, True, ""),
     (False, True, True, False, "")],
    [(True, False, False, False, ""),
     (False, False, True, True, ""),
     (True, True, False, False, ""),
     (False, False, True, True, "")],
    [(True, False, False, False, "SORTIE"),
     (True, False, True, False, ""),
     (True, False, True, True, ""),
     (False, False, True, False, "")]
    ]
```

Qu'on peut représenter graphiquement par :



3. et un troisième labyrinthe représenté graphiquement par :



Question 3. Donner une fonction `deplace_possible` qui prend en entrée un labyrinthe `la`, des coordonnées `c` qui coorespondent à une case de `la` et une direction `d` et qui décide (qui renvoie `True` quand c'est vrai

et `False` quand c'est faux) si le déplacement depuis la case de coordonnées `c` dans `la` dans la direction `d` est possible.

```
assert deplace_possible(laby1, (0, 1), "S")
assert not deplace_possible(laby1, (0, 1), "N")
assert not deplace_possible(laby1, (0, 1), "E")
```

Question 4. Donner une fonction **récursive** `chemin_possible` qui prend en entrée un labyrinthe `la`, des coordonnées `c` et un chemin `ch` et qui décide si l'itinéraire partant de la case de coordonnées `c` dans `la` et suivant le chemin `ch` est possible (c'est-à-dire, si à aucun moment on essaye de traverser un mur).

```
assert chemin_possible(laby1, (0, 1), ["S", "E", "N"])
assert chemin_possible(laby1, (0, 1), ["S", "N", "S", "E", "N"])
assert not chemin_possible(laby1, (0, 1), ["S", "O"])
assert not chemin_possible(laby1, (0, 1), ["S", "E", "N", "O"])
```

Question 5. Donner une fonction `est_solution` qui prend en entrée un labyrinthe `la`, des coordonnées `c` et un chemin `ch` et qui décide si oui ou non le couple (`c`, `ch`) est une solution du labyrinthe, c'est-à-dire si :

- `c` est une case d'entrée de `la`,
- le chemin `ch` est possible depuis `c` dans `la`,
- le chemin `ch` depuis `c` dans `la` arrive dans une case de sortie.

```
assert est_solution(laby1, (0, 1), ["S", "E", "N"])
assert est_solution(laby1, (0, 1), ["S", "E", "O", "E", "N"])
assert not est_solution(laby1, (0, 0), ["E", "N"])
assert not est_solution(laby1, (0, 1), ["S", "E"])
assert not est_solution(laby1, (0, 1), ["E"])
```

2 Suggestion : Marche aléatoire

On s'intéresse à la résolution d'un labyrinthe, c'est-à-dire à la recherche d'un chemin de l'entrée vers la sortie.

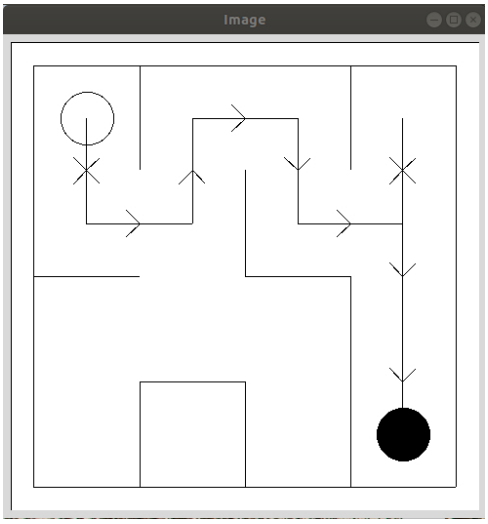
Une première solution consiste à errer au hasard dans le labyrinthe. A chaque étape :

- si on est sur une sortie, on s'arrête, on a gagné,
- sinon, on regarde quelles sont les directions de déplacement possibles depuis la case courante,
- on choisit une direction au hasard parmi les directions de déplacement possibles,
- on se déplace dans la case correspondante.

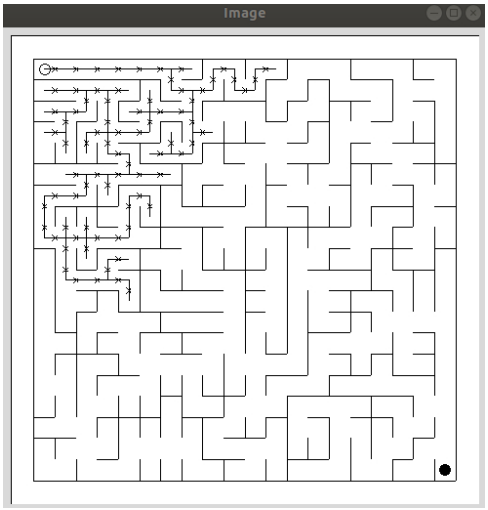
Question. Ecrire une fonction **récursive** qui erre au hasard dans un labyrinthe (on donnera une limite au nombre de déplacements possibles).

Par exemple pour `laby1` la fonction peut renvoyer `['S', 'E', 'O', 'E', 'N']`.

Pour `laby2`, voici la représentation d'un chemin aléatoire possible :



Pour `laby3`, voici la représentation d'un chemin de 900 étapes possible :



Remarque : En Python, la limite de profondeur de récursion est autour de 1000 appels, on flirte donc avec cette limite dans le cas de l'exemple précédent.

3 Suggestion : Main droite

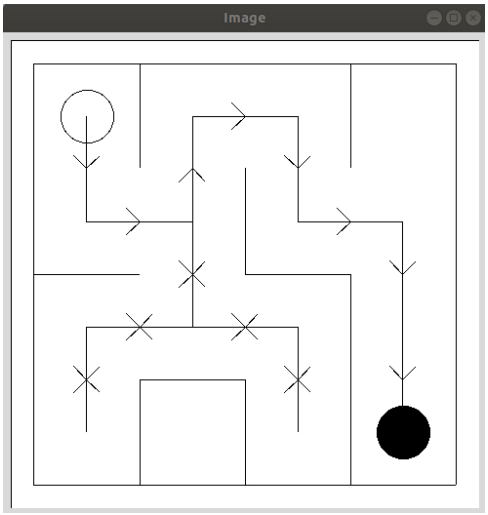
Une deuxième solution pour résoudre certains labyrinthes consiste à **toujours prendre le chemin le plus à droite possible** (on dit aussi qu'on "suit le mur de droite avec la main").

Par exemple, quand on arrive sur une case depuis la case située au Nord, on "regarde" vers le Sud, et on va commencer par vérifier si on peut prendre la direction Ouest (qui se situe à notre droite), si oui, on se déplace, sinon, on regarde la direction Sud, puis l'Ouest et enfin le Nord.

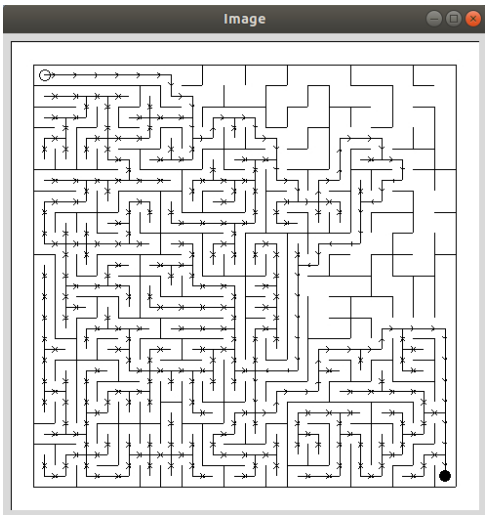
Question. Ecrire une fonction **récursive** qui explore le labyrinthe en tournant toujours à droite.

Par exemple pour `laby1` elle doit donner `['S', 'E', 'N']`.

Pour `laby2` elle doit donner le chemin représenté par :



Pour `laby3` elle doit donner le chemin représenté par :



4 Suggestion : Autres parcours

On pourra implémenter d'autres parcours de labyrinthes récursifs, notamment les parcours *en profondeur* et *en largeur*[2] (qui permet d'obtenir le chemin le plus court de l'entrée à la sortie).

5 Suggestion : Labyrinthes bien-formés

On pourra implémenter des tests sur les labyrinthes, pour vérifier que ceux-ci sont bien formés, en testant par exemple que:

- toutes les colonnes du labyrinthe ont le même nombre de cases,
- on ne peut pas "sortir des limites" du labyrinthe (i.e. il est entouré de murs),
- si on peut aller d'une case à sa voisine dans une direction, alors on peut aller de la voisine dans la case initiale dans la direction opposée,
- le labyrinthe a exactement une entrée et une sortie,
- la sortie est accessible depuis l'entrée, ...

[2] au programme de NSI de Terminale.

6 Suggestion : Affichage

En utilisant la bilbiothèque graphique du TME 1.9, on pourra écrire des fonctions `dessine_laby` et `dessine_chemin` qui produisent des images représentant, respectivement, des labyrinthes et des chemins dans un labyrinthes, comme vus sur les figures ci-dessus.

7 Suggestion : Génération

Une technique pour générer des labyrinthes intéressants et de commencer par créer un labyrinthe dans lequel chaque case est completement fermée (par quatre murs). Ensuite on répète les actions suivantes :

- on tire un mur au hasard dans le labyrinthe,
- on regarde si les deux cases situées de part et d'autre du mur sont joignables, c'est-à-dire s'il existe déjà un chemin de l'une à l'autre dans le labyrinthe
  - si c'est le cas, on recommence à tirer un mur,
  - sinon, on casse le mur qu'on a tiré.

On s'arrête quand chaque case est joignable depuis chaque autre case.

La difficulté de cette technique est la gestion de la "joignabilité" de deux cases. Une manière de procéder est de maintenir des "classes de joignabilité" : au départ, chaque case est seule dans sa classe, et quand on casse un mur, on réunit les classes des cases de part et d'autre du mur, qui deviennent une seule classe (on a donc une classe de moins, et on s'arrête quand on n'a plus qu'une classe). Pour tester si deux cases sont joignables, il suffit de vérifier si elles sont dans la même classe.

8 Suggestion : Jeu interactif

A la manière du *Livre dont vous êtes le héros* de l'activité 1, on peut proposer un jeu interactif d'exploration du labyrinthe, à chaque appel de la fonction de jeu, un texte décrit la pièce dans laquelle on se trouve, et les directions de déplacement possibles.