Un labyrinthe est dit « parfait » s'il existe un unique chemin permettant de relier n'importe quel couple de cases de ce labyrinthe.
Pour simplifier, on se limitera à des labyrinthes carrés (donc autant de lignes que de colonnes). Un labyrinthe sera modélisé par un graphe (non orienté) dont les sommets sont les coordonnées des cases `(i,j)`, où i et j sont des entiers de $[\![1, n]\!]$. Une arrête joindra deux sommets s'il est possible de passer de l'un à l'autre, donc si ces deux sommets sont voisins dans le labyrinthe et s'il n'y a pas de mur entre les deux cases qu'ils représentent. Les cases sont numérotées en partant d'en bas à gauche (la case en bas à gauche est associée au sommet `(0,0)`).
Le graphe représentant un labyrinthe sera modélisé par son dictionnaire d'adjacence : les clés sont les couples représentant les coordonnées des sommets et la valeur associée est la liste des sommets accessibles depuis cette case.
Pour commencer, récupérez sur le réseau, et copier sur votre répertoire de travail, le fichier *labEleves.py* qui contient les éléments suivants :

- Une fonction `affichage(lab:dict,chemin=[])` qui permet d'afficher un labyrinthe `lab` et le chemin permettant d'atteindre la sortie (la case en bas à gauche). Si la variable `chemin` n'est pas précisée, la fonction se contente d'afficher le labyrinthe.
- Un petit labyrinthe `labTest` (de taille 3×3) utile pour tester vos fonctions.
- Un labyrinthe `Lab`, plus grand (pour faire un joli dessin quand vous aurez fini la première partie).

Pour commencer, vous pouvez vérifier la forme des labyrinthes en utilisant `affichage(labTest)` et `affichage(Lab)`.

I Sortir d'un labyrinthe

Dans cette partie, on va programmer une fonction permettant de trouver un chemin permettant de relier une case quelconque du labyrinthe de coordonnées (i, j) $(0 \leqslant i, j \leqslant n-1)$ et la sortie (supposée en bas à gauche, case $(0, 0)$). On supposera que le labyrinthe est parfait, ce qui assure qu'un tel chemin existe (et est unique). Pour cela, on va adapter l'algorithme de parcours en profondeur d'un graphe : de façon à pouvoir reconstruire le chemin qui va vers la sortie, on introduira un dictionnaire `origine`, initialement vide, dont les clés sont les sommets visités `s` du graphe et la valeur associée est le sommet depuis lequel on a atteint `s`.

1. Écrire une fonction `exit(lab:dict,depart:tuple)->list` qui prend en argument un labyrinthe `lab`, les coordonnées d'une case `depart` (donc un couple d'entiers) et qui renvoie la liste des cases par lesquelles il faut passer pour rejoindre la sortie.
2. Tester votre fonction sur le labyrinthe `Lab` avec la fonction affichage ; le chemin devrait apparaître en pointillés rouges.

II Création d'un labyrinthe parfait

On va cette fois ci utiliser la même démarche pour créer un labyrinthe parfait. On va partir d'un labyrinthe de taille n dont toutes les cases sont fermées par 4 murs. Puis, en partant de la sortie, on va chercher à rejoindre toutes les autres cases. On va pour cela faire « tomber » certains murs.
Pour cela, on va faire un parcours « en profondeur » du graphe de la façon suivante : pour un sommet `s`, au lieu d'empiler les sommets accessibles depuis `s` (il n'y en a pas tant que les murs n'ont pas été ouverts), on va empiler les cases non visitées parmi les quatre cases voisines de `s` (si on est sur le bord du labyrinthe, il n'y en a que deux ou trois).
On pourra à nouveau utiliser un dictionnaire `origine`, construit de façon similaire à celui de la première partie, de façon à mémoriser le chemin fait dans le labyrinthe afin de créer le graphe final du labyrinthe : il suffira, à partir de la case de sortie `(0,0)`, de faire tomber tous les murs rencontrés lors de ce parcours.
On commence une fonction annexe :

1. Écrire une fonction `initialiser(n:int)->dict` qui prend en argument un entier `n` (non nul) et qui renvoie un labyrinthe de taille $n \times n$ dont toutes les cases sont fermées.

De façon à ne pas toujours créer le même labyrinthe, on va faire un tirage au sort pour choisir le mur à abattre : la fonction `acces(lab:dict,case:tuple)->list` renvoie la liste des sommets accessibles depuis le sommet `case`, dans un ordre aléatoire.

2. Écrire une fonction `maze(n:int)->dict` qui prend en argument un entier n (non nul) et renvoie un labyrinthe parfait de taille $n \times n$. Cette fonction doit dans un premier temps « parcourir » le labyrinthe (en utilisant `acces`) pour remplir le dictionnaire `origine` puis reconstruire le labyrinthe en suivant le trajet précédent. *Pensez à créer un graphe non orienté*
3. Tester cette fonction en affichant le labyrinthe créé.