

Génération de labyrinthes parfaits aléatoires

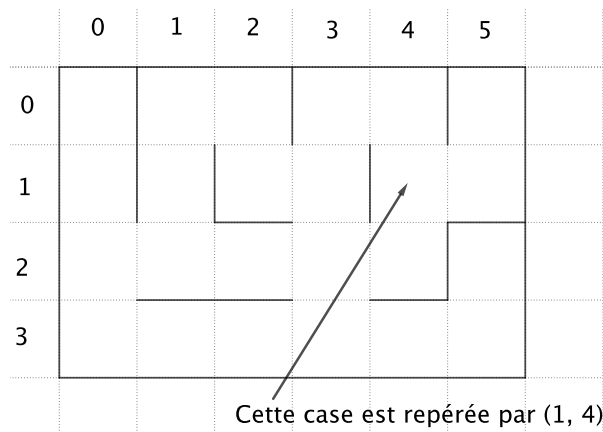
Attention, vous devez rendre votre code simultanément de deux manières :

- par mail à l'adresse *****@ac-lyon.fr au plus tard le dernier jour des vacances,
- rendre à la rentrée une copie manuscrite avec toutes les réponses (code compris) aux questions.

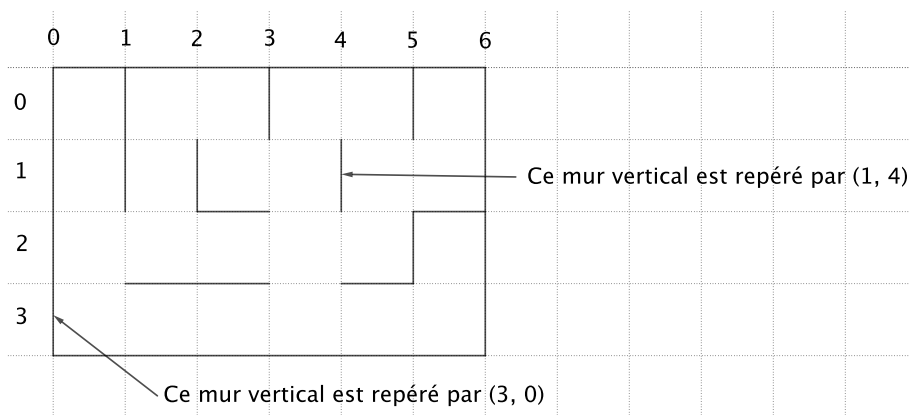
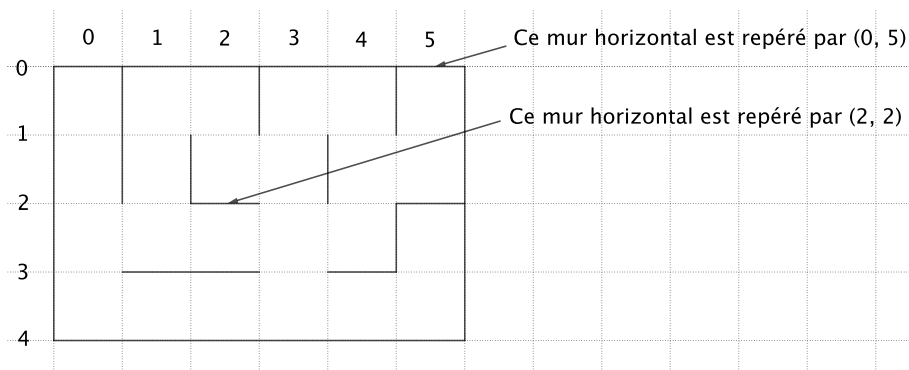
Quelques définitions

Un labyrinthe est un ensemble de cases ou cellules délimitées par des murs qui peuvent être présents ou pas. Les cases sont repérées par un couple formé de leur numéro de ligne et de colonne.

La dimension d'un labyrinthe est le couple (n, p) où n est le nombre de lignes et p le nombre de colonnes. Ci-contre, on a un labyrinthe de dimension $(4, 6)$.



Les murs horizontaux et les murs verticaux sont différenciés et on les repère de façon similaire :



On remarquera que, pour un labyrinthe de dimension (n, p) , les murs horizontaux sont repérés par des couples (i, j) avec $0 \leq i \leq n$ et $0 \leq j \leq p - 1$ et les murs verticaux sont repérés par des couples (i, j) avec $0 \leq i \leq n - 1$ et $0 \leq j \leq p$.

Partie A : Représentation des labyrinthes

Pour afficher les labyrinthes, on va simplement utiliser le module `turtle` mais il faut d'abord bien comprendre leur représentation.

Un labyrinthe est la donnée de deux ensembles : l'ensemble des murs verticaux et l'ensemble des murs horizontaux. Plus précisément, un labyrinthe de dimension (n, p) sera représenté par un couple $(\text{MursH}, \text{MursV})$ où `MursH` et `MursV` sont deux listes de listes de booléens indiquant pour chaque élément si le mur correspondant est présent ou non : le booléen `MursH[i][j]` a pour valeur `True` si et seulement si il y a un mur horizontal présent à la position (i, j) et le booléen `MursV[i][j]` indique de la même façon si un mur vertical est présent à la position (i, j) . Bien étudier l'exemple en identifiant pour chaque mur à quel booléen il correspond puis répondre à la question 1.

Le codage des murs horizontaux du labyrinthe ci-contre est donné par :

`[[True, True, True], [False, False, True], [True, True, True]]`

Et le codage des murs verticaux est donné par :

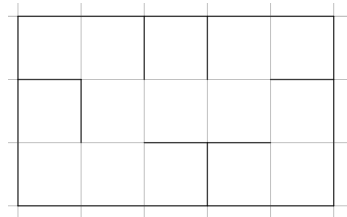
`[[True, True, False, True], [True, False, False, True]]`



1.

Pour le labyrinthe ci-contre, donner précisément les listes `MursH` et `MursV` qui le représentent.

Attention, bien traiter cette question est important pour la suite.



- Écrire une fonction `dim_laby(Laby)` qui renvoie la dimension du labyrinthe passé en paramètre. On rappelle que la dimension d'un labyrinthe est un couple d'entiers.
- Pour importer et configurer le module `turtle` et pour le tirage aléatoire, il vous faudra ajouter les lignes suivantes en tête de votre code :

```
1 from turtle import *
2 from random import randint
3
4 setworldcoordinates(-20, -20, 320, 320)
5 speed(0)
6 hideturtle()
```

On donne la première partie de la fonction qui permet d'afficher un labyrinthe : elle ne fait que l'affichage des murs horizontaux.

Lire le code, le comprendre et le compléter pour obtenir le code complet en ajoutant le traitement des murs verticaux. Attention, on rappelle que le système de coordonnées utilisé par `turtle` est celui qui vous a été enseigné en cours de math pour le repérage du plan : La fonction `goto(x, y)` déplace la tortue jusqu'à la position (x, y) , `up()` relève le stylo et `down()` l'abaisse.

```
1 def affiche_laby(Laby, Lmurs = 10):
2     (n, p) = dim_laby(Laby)
3     (MursH, MursV) = Laby
4
5     for i in range(n + 1):
6         M = (0, (n-i)*Lmurs)
7         up()
```

```

8      for j in range(p):
9          goto(M)
10         if MursH[i][j]:
11             down()
12         else:
13             up()
14             (x, y) = M
15             M = (x + Lmurs, y)
16             goto(M)
17
18 # à compléter

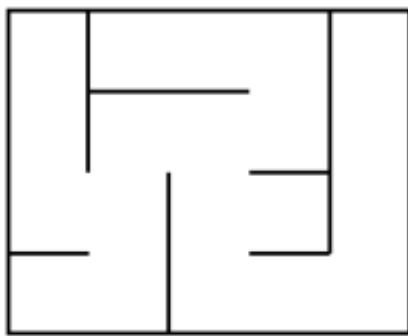
```

Remarque : En ajoutant `exitonclick()` à la fin de votre fonction, un clic sur la fenêtre `turtle` provoquera sa fermeture.

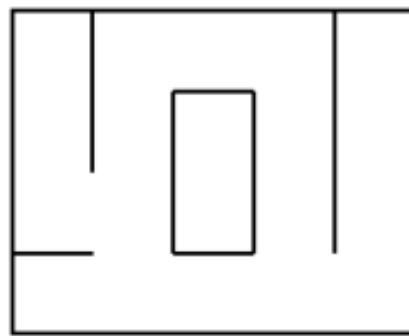
Partie B : Génération de labyrinthes parfaits

Dans un labyrinthe, un chemin est la donnée d'une suite de cases deux à deux distinctes telle que deux cases consécutives soient toujours adjacentes et non séparées par un mur. La première case s'appelle case départ et la dernière, case arrivée. Attention, dans un chemin, on ne peut pas revenir en arrière car les cases sont toutes différentes.

On dit alors qu'un labyrinthe est parfait lorsque deux cases différentes sont toujours reliées par un unique chemin :



Labyrinthe « parfait »



Labyrinthe « imparfait »

On va maintenant générer aléatoirement des labyrinthes parfaits par l'algorithme d'**exploration exhaustive** qui est exposée sur la page :

https://fr.wikipedia.org/wiki/Modélisation_mathématique_de_labyrinthe

Il est conseillé de bien étudier le paragraphe détaillant le fonctionnement de cet algorithme.

1. Compléter le code de la fonction booléenne suivante en respectant la spécification donnée dans la docstring

```

1 def position_valide(pos, dim):
2     """ Fonction booléenne qui teste si pos est un couple de coordonnées
3     d'une case dans les limites dim des dimensions d'un labyrinthe """
4
5     (n, p) = dim
6     (i, j) = pos
7     return # à compléter ici

```

2. On suppose que le labyrinthe est en cours de construction.

Vues est une liste de listes de booléens qui indique pour chaque case si elle a déjà été visitée : **Vues**[i][j] a pour valeur **False** si la case de coordonnées (i,j) n'a pas encore été visitée et **True** sinon.

Compléter le code de la fonction suivante en respectant la spécification donnée dans la docstring

```

1 def directions_possibles(Laby, Vues, pos):
2     """ Renvoie une liste de chaines de caractères indiquant les
3         différentes directions possibles pour désigner le mur
4         à abattre à partir de la position pos lors de l'exécution
5         de l'algorithme d'exploration exhaustive """
6
7     dim = dim_laby(Laby)
8     (MursH, MursV) = Laby
9     (i, j) = pos
10    directions = []
11
12    # vers le nord
13    nord = (i - 1, j)
14    if position_valide(nord, dim) and not Vues[i-1][j]:
15        directions.append('nord')
16
17    # vers le sud
18    # à compléter
19
20    # vers l'ouest
21    ouest = (i, j - 1)
22    if position_valide(ouest, dim) and not Vues[i][j-1]:
23        directions.append('ouest')
24
25    # vers l'est
26    # à compléter
27
28    return directions

```

3. On rappelle qu'un labyrinthe est la donnée d'un couple (**MursH**, **MursV**) comme défini à la partie A. Compléter le code de la fonction suivante en respectant la spécification donnée dans la docstring.

```

1 def abattre_mur(Laby, pos, direction):
2     """ Laby est un labyrinthe, pos désigne la position d'une
3         case et direction est une chaîne de caractère désignant le
4         mur à abattre du point de vue de cette case. Modifie Laby pour
5         changer la valeur du booléen qui représente le statut (présent
6         ou absent) du mur à abattre et renvoie la position de la case
7         vers laquelle on se déplace.
8         """
9
10    (MursH, MursV) = Laby
11    (i, j) = pos
12
13    if direction == 'nord':
14        MursH[i][j] = False
15        return (i - 1, j)
16    # à compléter

```

4. Pour générer un labyrinthe parfait en suivant l'algorithme, on va avoir besoin de conserver la liste des cases rencontrées au cours de l'exploration. C'est le rôle de la liste **Chemin** qui joue le rôle d'une pile. Détaillons l'algorithme :

Après avoir choisit la case qui sert de point de départ, on stocke sa position dans **Chemin** et on la marque comme vue.

Ensuite tant qu'il reste une case dans **Chemin**, on se place dans la dernière case :

- Si aucune direction n'est possible à partir de cette case pour poursuivre l'algorithme, on la retire de **Chemin**.
- Sinon, on choisit au hasard une des directions possibles puis on abat le mur correspondant, on marque la nouvelle case comme visitée dans **Vues** et on l'ajoute dans **Chemin**

Lorsque **Chemin** est vide c'est que l'algorithme est terminée.

Compléter le code de la fonction suivante en respectant la spécification donnée dans la docstring.

```

1 def genere_laby(n, p):
2     """ Renvoie un couple (MursH, MursV) qui représente un labyrinthe
3         aléatoire parfait de dimension (n, p) obtenu en appliquant
4         l'algorithme d'exploration exhaustive """
5
6     # tableau des cases visitées. Au départ, aucune n'a été visitée
7     Vues = [[False for _ in range(p)] for _ in range(n)]
8     # tableau des murs horizontaux, tous présents au départ
9     MursH = [[True for _ in range(p)] for _ in range(n + 1)]
10    # tableau des murs verticaux, tous présents au départ
11    MursV = [[True for _ in range(p + 1)] for _ in range(n)]
12    Laby = (MursH, MursV)
13
14    # On choisit au hasard le point de départ
15
16    (i, j) = (randint(0, n - 1), randint(0, p - 1))
17    depart = (i, j)
18    Chemin = [depart]
19    Vues[i][j] = True
20
21    while len(Chemin) > 0:
22        pos = Chemin[-1]
23        Directions = directions_possibles(Laby, Vues, pos)
24        if len(Directions) == 0: # Impasse
25            Chemin.pop()
26        else:
27            # à compléter
28            # avec autant de lignes
29            # que nécessaire
30
31    return Laby

```

