Corrigé

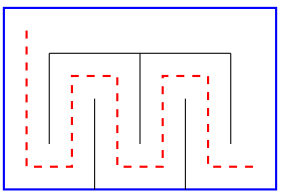
Parcours dans un labyrinthe

1. Parcours de labyrinthe

Question 1. Pour modéliser le fil d'Ariane un pile s'impose, dans laquelle nous allons empiler les coordonnées des cases parcourues ; chaque case qui entre dans la pile voit son attribut `etat` passer à `False`, puis on se dirige vers une case voisine dont l'attribut est encore égal à `True`. Dans le cas d'une impasse, la case est sortie de la pile, et on recommence avec le sommet suivant.

```
def explorer(laby):
    pile = Pile()
    pile.push((0, laby.q-1))
    laby.tab[0][laby.q-1].etat = False
    while True:
        i, j = pile.pop()
        if i == laby.p-1 and j == 0:
            break
        if j > 0 and laby.tab[i][j].S and laby.tab[i][j-1].etat:
            pile.push((i, j))
            pile.push((i, j-1))
            laby.tab[i][j-1].etat = False
        elif i < laby.p-1 and laby.tab[i][j].E and laby.tab[i+1][j].etat:
            pile.push((i, j))
            pile.push((i+1, j))
            laby.tab[i+1][j].etat = False
        elif j < laby.q-1 and laby.tab[i][j].N and laby.tab[i][j+1].etat:
            pile.push((i, j))
            pile.push((i, j+1))
            laby.tab[i][j+1].etat = False
        elif i > 0 and laby.tab[i][j].W and laby.tab[i-1][j].etat:
            pile.push((i, j))
            pile.push((i-1, j))
            laby.tab[i-1][j].etat = False
    return pile.lst
```

Question 2. Tel qu'il est écrit, le code explore préférentiellement les directions sud, est, nord et enfin ouest. Dans le labyrinthe présenté ci-dessous le chemin retourné (indiqué par des pointillés) n'est à l'évidence pas minimal :



Pour trouver le chemin minimal, il faudrait utiliser une file plutôt qu'une pile : à chaque fois qu'un élément sort de la file on fait entrer tous ses voisins non encore explorés. De la sorte toutes les cases à la distance 1 de l'entrée sont explorées, puis toutes celles à une distance 2, etc. On réalise ainsi une exploration *en largeur* du labyrinthe. Notons que contrairement à l'algorithme précédent, la file de garde pas trace du chemin parcouru et il faudrait donc mémoriser pour chaque case explorée le chemin qui y mène, engendrant ainsi un coût spatial beaucoup plus important.

2. Génération de labyrinthes

Question 3.

1. On définit les classes suivantes :

```
class Case:
    def __init__(self):
        self.N = False
        self.W = False
        self.S = False
        self.E = False
        self.etat = False
```

```
class Labyrinthe:
    def __init__(self, p, q):
        self.p = p
        self.q = q
        self.tab = [[Case() for j in range(q)] for i in range(p)]
```

2. Il y a une certaine redondance dans la modélisation des murs : si on ouvre un mur à l'est de la case de coordonnées (i, j), il faut aussi ouvrir un mur à l'ouest de la case de coordonnées $(i+1, j)$.

```
from numpy.random import randint

def creation(p, q):
    laby = Labyrinthe(p, q)
    pile = Pile()
    i, j = randint(p), randint(q)
    pile.push((i, j))
    laby.tab[i][j].etat = True
    while not pile.empty():
        i, j = pile.pop()
        v = []
        if j < q-1 and not laby.tab[i][j+1].etat:
            v.append('N')
        if i > 0 and not laby.tab[i-1][j].etat:
            v.append('W')
        if j > 0 and not laby.tab[i][j-1].etat:
            v.append('S')
        if i < p-1 and not laby.tab[i+1][j].etat:
            v.append('E')
        if len(v) > 1:
            pile.push((i, j))
        if len(v) > 0:
            c = v[randint(len(v))]
            if c == 'N':
                laby.tab[i][j].N = True
                laby.tab[i][j+1].S = True
                laby.tab[i][j+1].etat = True
                pile.push((i, j+1))
            elif c == 'W':
                laby.tab[i][j].W = True
                laby.tab[i-1][j].E = True
                laby.tab[i-1][j].etat = True
                pile.push((i-1, j))
            elif c == 'S':
                laby.tab[i][j].S = True
                laby.tab[i][j-1].N = True
                laby.tab[i][j-1].etat = True
                pile.push((i, j-1))
            else:
                laby.tab[i][j].E = True
                laby.tab[i+1][j].W = True
                laby.tab[i+1][j].etat = True
                pile.push((i+1, j))
    return laby
```

Dans la fonction ci-dessus, on détermine dans une liste v la liste des voisins non encore réunis au labyrinthe puis

on tire au hasard la direction que l'on prend parmi ces différentes possibilités, de manière à générer un labyrinthe raisonnablement « aléatoire ». On ne garde la case de coordonnées (i, j) dans la pile que si v contient au moins deux éléments.

3. Au début de l'algorithme, le labyrinthe possède pq composantes connexes ; chaque étape lui en fait perdre une donc on creuse exactement $pq-1$ murs avant d'en avoir fini. Initialement il y avait $p(q-1)+q(p-1)$ murs internes donc quand le labyrinthe est créé il en reste $pq-p-q+1=(p-1)(q-1)$.

Question 4. Pour tracer la grille du labyrinthe, on ajoute à la définition de la classe *Labyrinthe* la méthode suivante :

```
    def show(self):
        plt.plot([0, 0, self.p, self.p, 0], [0, self.q, self.q, 0, 0], linewidth=2)
        for i in range(self.p-1):
            for j in range(self.q):
                if not self.tab[i][j].E:
                    plt.plot([i+1, i+1], [j, j+1], 'b')
        for j in range(self.q-1):
            for i in range(self.p):
                if not self.tab[i][j].N:
                    plt.plot([i, i+1], [j+1, j+1], 'b')
        plt.axis([-1, self.p+1, -1, self.q+1])
        plt.show()
```

(Du fait de la redondance on ne trace que les murs est et nord.)
Pour afficher la solution d'un labyrinthe, on ajoute la méthode :

```
    def solution(self):
        sol = explorer(self)
        X, Y = [], []
        for (i, j) in sol:
            X.append(i+.5)
            Y.append(j+.5)
        X.append(self.p-.5)
        Y.append(.5)
        plt.plot(X, Y, 'r', linewidth=2)
        self.show()
```

Par exemple :

```
laby = creation(60, 30)
laby.solution()
```