REPUBLIQUE TUNISIENNE
Ministère de l'enseignement Supérieure, de la recherche scientifique

Institut Préparatoire aux Etudes d'Ingénieurs de Sfax

الجمهورية التونسية
وزارة التعليم العالي
والبحث العلمي

المعهد التحضيري للدراسات الهندسية
بصفاقس

Epreuve d'informatique

1er semestre AU : 2018 – 2019

Date : Novembre 2018 ; **Durée :** 1H ; **Nombre de page :** 3

*L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.*

*Le langage de programmation sera obligatoirement **Python**.*

*

* *

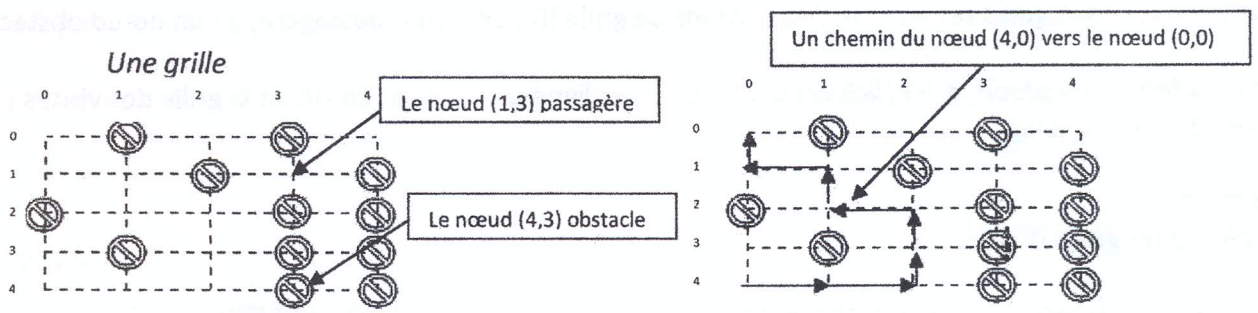
Implémentation. Dans ce sujet, nous adopterons la syntaxe du langage Python. On rappelle qu'en Python, il importe de bien respecter les indentations car elles permettent de définir des blocs.

Problème (Le labyrinthe)

Sur une grille discrète [0, n−1]×[0, n−1], on considère les n^2 nœuds c = (i, j). Deux nœuds de la grille sont voisins s'il existe une arête horizontale ou verticale les reliant. Un chemin entre deux nœuds c_1 et c_2 est la suite de nœuds successivement voisins $c_1, c_i, c_j, \ldots, c_k, c_2$, d'extrémités c_1 et c_2.

Un labyrinthe est tel que, pour une paire de nœuds de la grille, il existe un chemin entre ces nœuds sans passer par les nœuds obstacles ⊘. Le labyrinthe d'entrée c_1 et de sortie c_2 sera noté ici le chemin :
`[c1, ci]->[ci , cj ]->[cj , ck]->[ck, cr]->[cr , cp]->[cp , c2]`

Voici un exemple de labyrinthe de dimension 5 × 5, et un chemin qui entre du coin inférieur gauche (4, 0) et sort du coin supérieur gauche (0, 0).




```
Chemin =
[(4, 0), (4, 1)]->[(4, 1), (4, 2)]->[(4, 2), (3, 2)]->[(3, 2), (2, 2)]->[(2, 2),
(2, 1)]->[(2, 1), (1, 1)]->[(1, 1), (1, 0)]->[(1, 0), (0, 0)]
```

L'objectif de notre problème consiste, à partir d'une grille, trouver un chemin entre le nœud de départ et le nœud d'arrivée.

L'approche suivante profite de la structure de pile. On se donne :

— un **nœud** : un tuple (i,j) indiquant les coordonnés du nœud dans la grille.
— une **grilleDesVisites** : une liste de liste de booléens (1 :True, 0 :False), pour identifier les nœuds passagère : 0, les nœuds obstacles et déjà visités : 1 ,
— une pile **p** des nœuds non encore visités,
— une pile **laby** qui stocke le chemin du labyrinthe en construction.

On part du nœud d'entrée, par exemple (n-1, 0), que l'on empile sur **p** et que l'on marque dans **grilleDesVisites**.
Tant que la pile p n'est pas vide :

i) on dépile le "nœud-sommet" c de la pile p,
ii) on identifie ses nœuds voisins non encore visités et si il y en a au moins un :

- on en choisit un voisin aléatoirement : s,
- on empile le chemin [c, s] sur **laby**,
- on empile c puis s sur p,
- on marque s dans **grilleDesVisites** et on reprend.
- Si s est le nœud d'arrivée on s'arrête.

Les piles à traiter sont des piles bornées dont le premier élément indique la taille. On pourra utiliser les fonctions suivantes du **module pile** :

- Créer_pile_vide(c) : permet de créer une pile vide à capacité fini c
- Empiler(p, v) : ajouter v au sommet de la pile p
- Dépiler(p) : retire le sommet de la pile et retourne l'élément dépilé
- Sommet(p) : retourne le sommet de la pile.
- Est_vide(p) : vérifie si p est vide
- Taille(p) : renvoie la taille de p

Travail demandé :

Les questions suivantes vous accompagnent dans la mise en œuvre de cet algorithme. La seule difficulté est **d'éviter de sortir de la grille**.

Question1.
Les grilles de labyrinthe sont stockées dans des fichiers textes dont les lignes sont une suite de 0 et de 1 séparée par des virgules représentant les nœuds de grille (0 : un nœud passagère, 1 : un nœud obstacle).

Ecrire la fonction ***upload_grille(fichier)*** qui lit le fichier ligne par ligne et construit la grille des visites sous la forme d'une liste de liste.

Exemple :
Soit le fichier grille01.txt suivant :

grille01.txt

```
0,1,0,1,0
0,0,1,0,1
1,0,0,1,1
0,1,0,1,1
0,0,0,1,1
```

```
>>> grilleDesVisites = upload_grille('grille01.txt')
>>> print(grilleDesVisites)
[[0, 1, 0, 1, 0], [0, 0, 1, 0, 1], [1, 0, 0, 1, 1], [0, 1, 0, 1, 1], [0, 0, 0, 1, 1]]
```

Question2.

Ecrire la fonction ***initialiser_piles(n)*** qui prend en paramètre un entier n et qui crée et renvoie les deux piles bornées vides p et laby de capacité n^2.

Question3.

Ecrire la fonction visiter(noeud , grilleDesVisites) qui marque le nœud comme étant déjà visité dans la grille des visites.

Exemple :

```
>>> visiter( (3,2) , grilleDesVisites )
>>> print(grilleDesVisites)
[[0, 1, 0, 1, 0], [0, 0, 1, 0, 1], [1, 0, 1, 1, 1], [0, 1, 1, 1, 1], [0, 0, 0, 1, 1]]
```

Question 4.

Ecrire la fonction **dejaVisite(noeud, grilleDesVisites)** qui renvoie l'état de visite du nœud donné, si le nœud est dans la grille. Sinon (si le nœud est hors grille) la fonction renvoie 1.

Question 5.

Ecrire la fonction **voisins (nœud , grilleDesVisites)** qui renvoie la liste des nœuds voisins non encore visités du nœud donné (passé en paramètre).

Question6.

Ecrire la fonction **choisirVoisin(nœud , grilleDesVisites)** qui choisit et renvoie un nœud voisin au hasard non encore visité du nœud donné. Cette fonction traite l'exception dans le cas où le nœud ne possède pas de voisins.

NB : Utiliser la fonction choice(L) du module random qui renvoie au hasard un élément de la liste L.

Question 7.

Ecrire la fonction **labyrinthe(grilleDesVisites, depart , arrivee)** qui prend en paramètre la grille des visites, le nœud de départ et le nœud d'arrivée. Cette fonction calcule et renvoie la pile **laby** selon l'algorithme décrit précédemment.

Exemple :

```
>>> laby = labyrinthe(grilleDesVisites, (4,0) , (0,0) )
>>> print(laby)
[8, [(4, 0), (4, 1)], [(4, 1), (4, 2)], [(4, 2), (3, 2)], [(3, 2), (2, 2)],
[(2, 2), (2, 1)], [(2, 1), (1, 1)], [(1, 1), (1, 0)], [(1, 0), (0, 0)],
None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None]
```

None
...
None
[(1, 0), (0, 0)]
[(1, 1), (1, 0)]
[(2, 1), (1, 1)]
[(2, 2), (2, 1)]
[(3, 2), (2, 2)]
[(4, 2), (3, 2)]
[(4, 1), (4, 2)]
[(4, 0), (4, 1)]
8

Pile borne : laby

Question 8.

Ecrire la fonction **tracer_chemin(laby)**qui parcourt la pile laby et renvoie le chemin du labyrinthe en chaine de caractères sous la forme ' $[c_1, c_i]$->$[c_i , c_j]$->$[c_j , c_k]$->$[c_k, c_r]$->$[c_r , c_p]$->$[c_p , c_2]$ ' .

Exemple :

```
>>> chemin = tracer_chemin(laby)
>>> print(chemin)
' [(4, 0), (4, 1)]->[(4, 1), (4, 2)]->[(4, 2), (3, 2)]->[(3, 2), (2, 2)]->[(2, 2), (2, 1)]->[(2, 1), (1, 1)]
->[(1, 1), (1, 0)]->[(1, 0), (0, 0)] '
```

En déduire la **complexité** au pire des cas de la fonction **tracer_chemin** en fonction de n.