UFR de mathématique et d'informatique
Université de Strasbourg

Licence 1 Semestre 1 Automne
Algorithmique et Programmation 1
Contrôle continu
Vendredi 21 décembre 2018
Durée : 1 heure 45
Cours et documents autorisés sauf le web.

Les exercices de cette épreuve ne sont pas liés au projet labyrinthe mais, si vous avez réussi le projet labyrinthe, vous parviendrez facilement à répondre à toutes les questions. Seule la dernière question de l'épreuve est très différente de ce que vous avez fait dans le projet labyrinthe.

Dans cette épreuve, vous ne pourrez pas utiliser les fonctions et méthodes des librairies *python*, sauf `range` et `len` ou sauf lorsque vous y êtes invités par l'énoncé. Par contre si, pour la réalisation d'une tâche, vous avez besoin d'écrire des fonctions qui ne sont pas explicitement demandées par l'énoncé, sentez-vous libre de le faire.

Toutes vos fonctions devront être testées (avec un *doctest* ou non). Nous vous invitons à écrire, à la suite de chaque fonction, les instructions que vous avez lancées pour tester cette fonction. Une fois les tests terminés, vous pouvez mettre ces instructions en commentaire si vous le souhaitez.

Nous vous invitons à porter une attention particulière au profil des fonctions que nous vous demandons d'écrire. Par exemple, une fonction avec deux paramètres entiers, n'est pas du tout équivalente à une fonction avec un seul paramètre constitué d'un couple d'entiers.

Nous vous demandons d'écrire toutes vos fonctions dans un même fichier.

Le barême de chaque question est donné à titre indicatif.

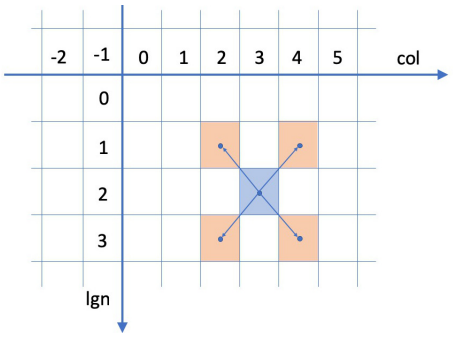
Parcours aléatoire d'un pion

Plaçons-nous dans le contexte d'un jeu où le ou les pions se déplacent sur un plateau constitué de *cellules* ou *cases* (comme sur un damier). On se limite au cas où il y a un seul pion. On souhaite que ce pion se déplace de façon aléatoire.

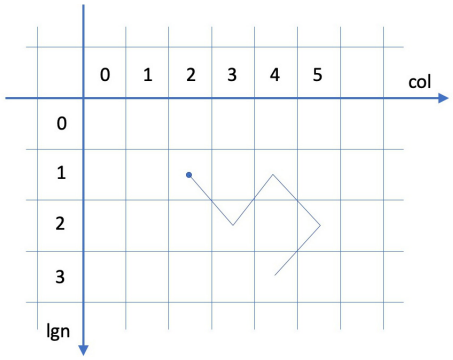
Chaque cellule est caractérisée par deux entiers représentant respectivement le numéro de la ligne et le numéro de la colonne où elle se trouve. Le *mouvement* du pion sera représenté par la liste de ses positions au cours du temps.

Pour chaque cellule, on peut définir des cellules *voisines*. Mais le voisinage n'est pas défini comme dans le projet labyrinthe. Ici deux cellules sont voisines si et seulement si la distance de l'une à l'autre est de $\sqrt{2}$. En pratique les *voisines* d'une cellule sont celles qui la touchent par les coins. Dans la figure 1a, on peut voir que la cellule (2,3) a pour voisin les cellules (1,2), (1,4), (3,4) et (3,2).

Le pion peut se déplacer d'une cellule à l'une de ses voisines choisies au hasard. Après plusieurs déplacements, on obtient un trajectoire aléatoire (cf figure 1b).



(a) La cellule (2,3) et ses 4 voisines : (1,2), (1,4), (3,4) et (3,2).



(b) L'objectif est de créer un parcours aléatoire en allant d'une cellule à l'une de ses voisines choisies au hasard.

1 Préparations (5 points = 0.5+0.5+1+1+2)

Dans cette section, vous pouvez utiliser la fonction `random.randint` du module `random` qui, à partir de deux entiers `n` et `m` (avec `n<=m`) retourne un entier choisi au hasard dans l'intervalle `[n,m]`. Par exemple `random.randint(2,5)` retourne, au hasard, 2, 3, 4 ou 5 avec la même probabilité. Vous pouvez aussi, si vous le souhaitez, utiliser la fonction `random.random()` qui retourne un nombre flottant pris au hasard dans l'intervalle `[0,1[`.

1. Importer le module `random`.
2. Écrire une fonction `entre0etN` qui, à partir d'un entier `n` retourne un entier au hasard entre `0` et `n`.
3. Écrire une fonction `elementAuHasard` qui, à partir d'une liste `lst` retourne au hasard un des éléments de cette liste. Par exemple si `lst=[2,0,20]`, alors `elementAuHasard(lst)` retourne un nombre choisi au hasard entre 2, 0 et 20.
4. Écrire une fonction `listeZeros` qui, à partir d'un entier `nb_elems` retourne une liste composée de `nb_elems` elements nuls. Par exemple `listeZeros(5)` retourne `[0,0,0,0,0]`.
5. Écrire une fonction `plateauZeros` qui, à partir de deux entiers `nb_lignes` et `nb_colonnes` retourne une liste composée de `nb_lignes` listes, chacune composée de `nb_colonnes` entiers nuls. Par exemple `plateauZeros(3,2)` retourne `[ [0,0], [0,0], [0,0] ]`.

Dans toute la suite, les cellules du plateau, ainsi que la position des pions sont représentés par des couples d'entiers. Dans ce couple, le premier élément représente le numéro de la ligne et le second élément le numéro de la colonne.

2 Plateau infini (PI) (5 points = 1+2+2)

Dans un premier temps, on suppose que le plateau sur lequel évoluent les pions est infini. Autrement dit, quelque soit la valeur des entiers `lgn` et `col`, le couple `(lgn,col)` représente toujours une cellule du plateau.

1. Écrire une fonction `voisines_PI` qui, à partir d'une cellule (couple d'entiers) `cell` retourne la liste des cellules voisines de `cell`. Par exemple, comme mentionné plus haut, pour `cell=(2,3)`, l'appel `voisines_PI(cell)` retourne la liste `[(1,2), (1,4), (3,4), (3,2)]` (cf figure 1a).
2. Écrire une fonction `voisine_PI_alea` qui, à partir d'une cellule `cell`, retourne une cellule choisie au hasard parmi toutes les voisines de `cell`.
3. Écrire une fonction `afficheParcoursNpas_PI` qui, à partir d'un entier `nb_pas` et d'une cellule `cell`, affiche `cell`, puis une cellule voisine de `cell` choisie au hasard, puis une voisine de la voisine choisie au hasard et ainsi de suite jusqu'à avoir effectué `nb_pas` pas (et avoir affiché `1+nb_pas` cellules). Il n'est pas nécessaire que toutes les voisines ainsi trouvées soient différentes. Par exemple pour `cell=(1,2)`, l'appel `afficheParcoursNpas_PI(8,cell)` pourrait afficher quelque chose comme la sortie ci-contre, ce qui représente le parcours représenté sur la figure 1b.

```
(1,2)
(2,3)
(1,4)
(2,5)
(3,4)
```

3 Plateau fini (PF) (3 points = 1+1+1)

À présent, on suppose que le plateau sur lequel évoluent les pions possède `nb_lignes` lignes et `nb_colonnes` colonnes. Autrement dit, `(lgn,col)` représente une cellule du plateau seulement si `lgn` appartient à `[0,nb_lignes-1]` et `col` appartient à `[0,nb_colonnes-1]`.

1. Écrire une fonction `estSurPlateau` qui, à partir d'une cellule `cell`, et des dimensions du plateau `nb_lignes` et `nb_colonnes` retourne un booléen valant `True` si `cell` est sur le plateau et `False` s'il n'y est pas.

2. Écrire une fonction `voisines_PF` qui, à partir d'une cellule `cell` et des dimensions du plateau `nb_lignes` et `nb_colonnes`, retourne la liste des cellules voisines de `cell` et appartenant au plateau. Vous pouvez utiliser `voisines_PI` et `estSurPlateau`.
3. Écrire une fonction `parcoursNbPas_PF` qui, à partir d'un entier `nb_pas`, d'une cellule de départ `cell` et des dimensions du plateau, retourne une liste de cellules commençant par `cell`, puis une voisine de `cell` choisie au hasard, puis une voisine de cette voisine et ainsi de suite, jusqu'à avoir trouvé `nb_pas` voisines. La liste retournée devra contenir `1+nb_pas` éléments et aucune des cellules ne doit sortir du plateau.

4 Éviter les obstacles (5 points = 0.5+1.5+1+1+1)

1. En utilisant la fonction `plateauZeros`, créer une liste appelée `obstacles` composée de `nb_lignes` listes, chacune composée de `nb_colonnes` entiers nuls. Pour cela, vous pouvez utiliser la fonction `plateauZeros`.
2. Parcourir les lignes et les colonnes de cette liste et transformer aléatoirement 40% de ces éléments en 1. Pour cela, vous pouvez utiliser la fonction `random.random()`. Les cellules pour lesquelles `obstacle` n'est pas nul représentent des obstacles.
3. Écrire une fonction `voisines_libres` qui, à partir d'une cellule `cell`, et de la liste `obstacles`, en déduit les dimensions du plateau et retourne la liste des cellules voisines de `cell`, qui soient sur le plateau et qui ne soient pas des obstacles.
4. Écrire une fonction `voisine_libre_alea` qui, à partir d'une cellule `cell`, et de la liste `obstacles`, établit la liste des voisines libres (en appelant la fonction précédente). Si cette liste est vide, alors la fonction devra retourner `None`, sinon, elle devra retourner un élément de cette liste choisie au hasard.
5. Écrire une fonction `parcours_obst` qui, à partir d'un entier `nb_pas`, d'une cellule `cell` représentant un point de départ, et de la liste `obstacles`, retourne une liste de cellules commençant par `cell` et continuant par une voisine libre de `cell` choisie au hasard, puis par une voisine libre de cette voisine libre et ainsi de suite jusqu'à avoir trouvé `nb_pas` voisines.

5 Le serpent (2 points)

Dans cette section, on souhaite que le pion ne puisse plus aller sur des cellules qu'il a déjà occupées (comme s'il s'agissait d'un serpent). Pour cela, on utilise le même mécanisme que ci-dessus (liste `obstacles`), mais cette liste est initialisée à 0 pour toutes les cellules. Autrement dit, au départ, il n'y a aucun obstacle. Mais, à chaque fois qu'une cellule `c` est visitée, on modifie la liste `obstacles` de façon à ce que `c` devienne un obstacle.

Écrire une fonction récursive `parcoursSerpent` qui, à partir d'une cellule de départ `cell` et d'une liste `obstacle` retourne, sous la forme d'une liste de cellules, un parcours aléatoire où toutes les cellules consécutives sont voisines et sont sur le plateau et où toutes les cellules sont distinctes. Cette fonction ne s'arrête pas après un certain nombre de pas, mais lorsque la cellule courante n'a plus de voisine libre. Voici une proposition d'algorithme :

— Si toutes les cellules voisines de `cell` sont des obstacles (ont déjà été visitées) alors l'algorithme s'arrête et retourne une liste composée seulement de `cell` ;

— sinon, soit `cell2` une voisine de `cell` choisie au hasard parmi les voisines accessibles et non encore visitées. Faire de `cell` un obstacle et exécuter tout le présent algorithme en partant de `cell2`. Soit `serp` la liste retournée par cet algorithme. Retourner la liste obtenue en ajoutant `cell` en tête de `serp`.

Si votre fonction n'est pas récursive, elle n'obtiendra pas les deux points.

Lorsque vous avez fini votre travail, je vous demande de remettre votre travail sur *moodle* sur le lien associé à votre groupe.