LYCÉE CLEMENCEAU

PREMIER SEMESTRE, 2014-2015

MP / MP* - Informatique pour tou(te)s

Devoir Surveillé

Durée : cent vingt minutes

CONSIGNES: Vous justifierez vos réponses avec le plus grand soin. N'oubliez pas qu'il serait maladroit de copier sur votre voisin(e) qui a, de toute façon, écrit n'importe quoi. Aucune calculatrice ni tout autre engin relié à un quelconque réseau n'est autorisé.
Vous choisirez UN des deux sujets proposés en indiquant clairement votre choix en début de copie.
Votre note dépendra en grande partie de la clarté de votre rédaction. Vos codes doivent être correctement indentés, commentés, présentés. Vous choisirez des noms de variables évocateurs. Vous utiliserez le langage Python.

SUJET 1. *(X / ENS Cachan / ESPCI - 2013)*

Points fixes de fonctions à domaine fini

Dans ce problème, on s'intéresse aux points fixes des fonctions $f : E \to E$ où E est un ensemble fini.

On désigne par E_n l'ensemble $\{0, 1, ..., n-1\}$. On représente une fonction $f : E_n \to E_n$ par un tableau `t` de taille n tel que $f(x) =$ `t[x]` pour tout x de E_n.

Par exemple, f_0 qui à $x \in E_{10}$ associe $2x + 1[10]$ est représentée par :

1	3	5	7	9	1	3	5	7	9

.

On note f^k l'itérée k-ième de f.

Partie I. Recherche de point fixe : cas général

1. Écrire une fonction `admet_point_fixe(t)` qui prend en argument un tableau `t` et renvoie `True` si la fonction représentée par `t` admet un point fixe et `False` sinon.
Par exemple, avec le tableau de la fonction f_0 donnée en exemple, la fonction renverra `True`.

2. Écrire une fonction `nb_points_fixes(t)` qui prend en argument un tableau `t` et renvoie le nombre de points fixes de f représentée par `t`.

3. Écrire une fonction `itere(t,x,k)` qui prend en premier argument un tableau `t`, en second un entier de E_n et en troisième un entier naturel k et renvoie $f^k(x)$.

4. Écrire une fonction `nb_points_fixes_iteres(t,k)` qui prend en argument un tableau `t`, en second un entier naturel k et renvoie le nombre de points fixes de k.

Un élément z de E_n est dit *attracteur principal* de f si, et seulement si, z est un point fixe de f et pour tout x de E_n, il existe un entier $k \geqslant 0$ tel que $f^k(x) = z$.
Afin d'illustrer cette notion, on pourra vérifier que la fonction f_1 représentée par le tableau suivant admet 2 comme attracteur principal :

5	5	2	2	0	2	2

En revanche, on notera que la fonction f_0 donnée en introduction n'admet pas d'attracteur principal.

5. Écrire une fonction `liste_points_fixes(t)` qui renvoie la liste des points fixes de f représentée par `t`.
Écrire ensuite une fonction `admet_attracteur_principal(t)` qui prend en argument un tableau `t` et renvoie `True` si, et seulement si, la fonction f admet un attracteur principal et `False` sinon.

Le *temps de convergence* de f en $x \in E_n$ est le plus petit entier $k \geqslant 0$ tel que $f^k(x)$ soit un point fixe de f.
Pour la fonction f_1, le temps de convergence en 4 est 3 car $f_1(4) = 0$, $f_1^2(4) = 5$, $f_1^3(4) = 2$ et 2 est un point fixe.

6. Écrire une fonction `temps_de_convergence(t,x)` qui prend en premier argument un tableau `t` représentant une fonction f et en second un entier x de E_n et qui renvoie le temps de convergence de f en x.

7. Écrire une fonction `temps_de_convergence_max(t)` qui prend en argument un tableau `t` de taille n représentant une fonction f et renvoie le maximum des temps de convergence de ses éléments. **On impose un temps de calcul LINÉAIRE** en la taille n du tableau.

Joker : on pourra utiliser le principe de *mémoïsation* en introduisant un dictionnaire dont les clés sont les éléments de E_n et les valeurs leur temps de convergence.

Rappel : en Python, la syntaxe de construction des dictionnaires est `{clé:valeur}` et `dic[clé]` renvoie la valeur associée à la clé `clé`.

8. Justifier que la complexité de cette fonction est en $O(n)$.

Partie II. Recherche efficace de points fixes.

Toute fonction `point_fixe(t)` retournant un point fixe d'une fonction arbitraire est de complexité au mieux linéaire. On s'intéresse maintenant à des améliorations possibles de cette complexité lorsque la fonction considérée possède certaines propriétés spécifiques.

Premier cas

On considère le cas d'une **fonction croissante** de E_n dans E_n.

9. Écrire un prédicat `est_croissante(t)` avec `t` un tableau associé à une fonction $f : E_n \to E_n$.
On impose un temps de calcul de complexité `linéaire` en la taille du tableau. On justifiera cette complexité.

10. Écrire une fonction `point_fixe(t)` avec `t` représentant une fonction croissante $f : E_n \to E_n$. Cette fonction retourne un entier x de E_n tel que $f(x) = x$.
On impose un temps de calcul de complexité **logarithmique**.

11. Démontrer que la fonction termine et justifier que sa complexité est logarithmique.

Deuxième cas

Soit $\preceq$ une relation d'ordre (pas forcément totale...) sur un ensemble E. Une fonction $f : E \to E$ est *croissante au sens de* $\preceq$ si, et seulement si, pour tous x et y de E : $x \preceq y \implies f(x) \preceq f(y)$.
On dit qu'un élément m de E est un *plus petit élément* de E au sens de $\preceq$ si, et seulement si, pour tout x de E, $m \preceq x$.
On admet que pour tout ensemble fini E muni d'une relation d'ordre $\preceq$ et qui admet un plus petit élément au sens de $\preceq$, il existe un entier naturel k tel que $f^k(m)$ est un point fixe de f dans E.

12. Soit E un ensemble fini quelconque muni d'une relation d'ordre $\preceq$ et admettant un plus petit élément m au sens de $\preceq$. Soit $f : E \to E$ une fonction croissante au sens de $\preceq$ et soit k un entier naturel tel que $f^k(m)$ soit un point fixe de f dans E.
Démontrer qur $f^k(m)$ est en fait le plus petit point fixe de f au sens de $\preceq$.

Nous nous intéressons maintenant à un choix particulier d'ordre $\preceq$ appelé *ordre de divisibilité* et noté $|$. On note $a|b$ la relation d'ordre « a divise b » sur les entiers positifs.
Par exemple, la fonction représentée par le tableau ci-dessous est croissante au sens de l'ordre de divisibilité :

0	2	4	6	4	8	0	2	0	6

On remarque que, par la question précédente, toute fonction de E_n dans lui-même, croissante au sens de la divisibilité, a un plus petit point fixe au sens de la divisibilité.
On étend la définition du PGCD à des entiers naturels quelconques en convenant de définir le PGCD d'un entier $x \geqslant 0$ et de 0 comme valant x.

13. Soit f une fonction de E_n dans E_n croissante au sens de la divisiblité et notons $x_1, ..., x_m$ les points fixes de f dans E_n. Montrer que le plus petit point fixe de f au sens de la divisibilité est exactement le PGCD de $x_1, ..., x_m$.

14. Écrire une fonction `pgcd_points_fixes(t)` qui prend en argument un tableau représentant une fonction croissante au sens de la divisibilité et renvoie le PGCD de ses points fixes. On impose un temps de calcul **logarithmique** en n.

15. Justifiez que la fonction précédente est de complexité $O(\log(n))$.

SUJET 2.

Banque PT

1. Soit un entier naturel non nul n et une liste `t` de longueur n dont les termes valent 0 ou 1. Le but de cette question est de trouver le nombre maximal de 0 contigus dans `t` . Par exemple, le nombre de zéros contigus de la liste `t1` suivante vaut 4 :

`i`	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`t1[i]`	0	1	1	1	0	0	0	1	0	1	1	0	0	0	0

a. Écrire une fonction `nombreZeros(t,i)` prenant en paramètres une liste `t` de longueur n et un indice `i` compris entre 0 et $n-1$ et renvoyant :

$$\begin{cases} 0, \text{ si } t[i] = 1 \\ \text{le nombre de zéros consécutifs dans } t \text{ à partir de } t[i] \text{ inclus, si } t[i] = 0 \end{cases}$$

b. Comment obtenir le nombre maximal de zéros contigus d'une liste `t` connaissant la liste des `nombreZeros(t,i)` pour $0 \leqslant i \leqslant n-1$?
En déduire une fonction `nombreZerosMax(t)` renvoyant le nombre maximal de 0 contigus d'une liste `t` non vide. On utilisera la fonction `nombreZeros` .

c. Quelle est la complexité de la fonction `nombreZerosMax` construite à la question précédente ?

d. Trouver un moyen simple, toujours en employant `nombreZeros` , d'obtenir un algorithme plus performant.

2. Soit N un entier naturel non nul. On cherche à trier une liste L d'entiers naturels strictement inférieurs à N.

a. Écrire une fonction `comptage` d'arguments L et N, renvoyant une liste P dont le k-ième élément désigne le nombre d'occurences de l'entier k dans la liste L.

b. Utiliser la liste P pour en déduire une fonction `tri` , d'arguments L et N, renvoyant la liste L triée dans l'ordre croissant.

c. Donner la trace de la fonction `tri` sur une liste de 10 entiers inférieurs ou égaux à 5.
Comment effecteur ce test sur machine.

d. Quelle est la complexité temporelle de cet algorithme ? La comparer à la complexité d'un tri par insertion ou d'un tri fusion.

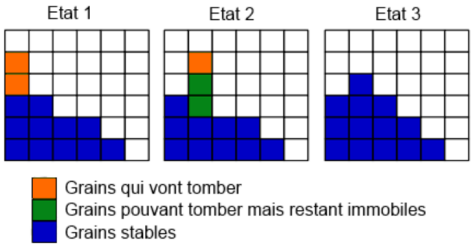
Mines/Ponts

Présentation des milieux granulaires

Le sable c'est cool, y'en a dans les dunes et on peut faire des sabliers avec. Le sable, ça fait des tas et on peut essayer de simuler l'évolution d'un tas numériquement. (Ceci condense l'introduction donnée dans le vrai sujet sur toute une page...)

Calcul de la forme d'un tas par automates cellulaires

L'objet de cette partie est la simulation numérique par un modèle de type automate cellulaire. Un automate cellulaire se présente généralement sous la forme d'un quadrillage dont chaque case peut être occupée ou vide. Un grain y est symbolisé par une case occupée. La configuration des cases, qu'on appelle état de l'automate, évolue au cours du temps selon certaines règles très simples permettant de reproduire des comportements extrêmement complexes. La physique n'intervient pas directement mais les règles d'évolution sont choisies de façon à reproduire au mieux les lois naturelles. Dans ce qui suit, nous allons simuler la formation d'un demi-tas de sable situé à droite de l'axe de symétrie vertical en appliquant les règles énoncées ci-après.



La représentation graphique est une image en deux dimensions mais l'automate est à une dimension car on considère le tas comme un ensemble de piles dont la hauteur future dépend uniquement des piles adjacentes : une cellule de l'automate cellulaire, c'est une pile. Le tas est complètement défini avec un tableau (ou une liste) unidimensionnel qui permet de stocker la hauteur des différentes piles. Une tour de hauteur h est une pile de cases pleines consécutives dont h voisines de droite sont vides. Si $h > 1$, on détermine arbitrairement le nombre n de grains du sommet de la tour qui vont tomber avec l'expression Python :

```
int( ( h + 2.0 )/2 * random())+ 1
```

Dans l'exemple figure 1, lors du passage de l'état 1 à l'état 2, le hasard introduit par la fonction `random()` (qui renvoie de manière aléatoire un flottant dans l'intervalle semi-ouvert [0.0, 1.0[) fait que toute la tour se décale apparemment vers la droite. Ensuite, pour le passage à l'état 3 seul un grain sur les trois possibles tombe. L'état 3 est stable, plus aucun grain ne tombe.

a. Quel est le type de l'expression `int( ( h + 2.0 )/ 2 * random())+ 1` ?
Déterminer un encadrement de sa valeur, sachant que $h > 1$.

b. Écrire une fonction nommée `calcul_n` qui prend la hauteur d'une pile comme argument et qui renvoie le nombre de grains qui vont tomber sur la pile suivante.
Il est rappelé que le (demi-)tas est complètement défini avec un tableau (ou une liste) unidimensionnel qui permet de stocker la hauteur des différentes piles. Au départ le support est vide et on vient déposer périodiquement un grain sur la première pile (à gauche) qui correspondra au sommet du demi-tas. Le support peut recevoir P piles et à son extrémité droite il n'y a rien, les grains tombent et sont perdus. La pile $P + 1$ est donc toujours vide.

c. Définir la fonction `initialisation` prenant un entier P en paramètre et renvoyant une liste contenant P piles de hauteur 0.

d. Définir une fonction `actualise` prenant comme paramètres une liste `piles` et un entier `perdus` qui va parcourir les piles de gauche à droite et les faire évoluer en utilisant les règles, fonctions et variables définies précédemment. Cette fonction doit renvoyer un couple formé d'une nouvelle liste décrivant la situation du tas de sable après ces opérations ainsi que du nombre de grains perdus (en prenant en compte ceux qui seront tombés de la dernière pile P).
Il est rappelé que Python permet de renvoyer un couple de valeurs A et B par l'instruction `return A, B`. On pourra, dans les questions suivantes, utiliser la fonction `actualise` par un appel de la forme suivante :

```
(nouveau_tas,nouveaux_perdus)= actualise(ancien_tas, anciens_perdus)
```

e. Écrire le bloc d'instructions du programme principal qui permet d'ajouter 1 grain à la première pile après chaque dizaine d'exécutions de la fonction `actualise` et qui s'arrêtera lorsqu'au moins 1000 grains seront sortis du support. Le nombre de piles (taille du support P) sera demandé à l'utilisateur lors de l'exécution. Vous utiliserez les fonctions et variables définies précédemment.