X Informatique PC 2013

Recherche de point fixe : cas général

1 Il s'agit ici de parcourir le tableau et de retourner `vrai` dès que l'on rencontre un point fixe. `faux` est retourné si l'on a parcouru l'ensemble du tableau sans trouver de point fixe. Une boucle `for` est utilisée pour parcourir le tableau et l'on met à profit le fait qu'une instruction `return` arrête immédiatement la procédure.

```
def admet_point_fixe(t):
    for i in range(len(t)):
        if t[i]==i:
            return True
    return False
```

2 Cette question est très similaire à la précédente, la différence étant qu'au lieu de retourner `vrai` lorsque l'on rencontre un point fixe, un compteur `pts_fixes`, initialisé à 0, est incrémenté.

```
def nb_points_fixes(t):
    p=0
    for i in range(len(t)):
        if t[i]==i:
            p+=1
    return p
```

3 Traitons cette question avec un code récursif. En effet, les fonctions f^k vérifient la relation de récurrence : pour tout $x \in \mathsf{E}_n$

$$f^0(x) = x \qquad \text{et} \qquad \forall k \geq 1, \quad f^{k+1}(x) = f(f^k(x))$$

La première ligne représente la condition d'arrêt du code.

```
def itere(t,x,k):
    if k==0:
        return x
    else:
        return itere(t,t[x],k-1)
```

Remarque 1

On peut aussi écrire un code itératif en appliquant k fois la fonction f, c'est-à-dire le tableau `t`, à l'aide d'une boucle `for`.

4 Modifions le code de la procédure `nb_point_fixes` de la question 2 en remplaçant le test `t[i]` = `i` par le même test mais sur l'itérée : `itere(t,i,k)` = `i`.

```
def nb_points_fixes_iteres(t,k):
    p=0
    for i in range(len(t)):
        if itere(t,i,k)==i:
            p+=1
    return p
```

5 Notons que si f admet un attracteur principal z, alors c'est son unique point fixe ; en effet, si y est un point fixe différent de z, on a $f^k(y) = y \neq z$ pour tout k, ce qui contredit l'hypothèse.

Comme l'énoncé ne demande pas de solution efficace à cette question, testons simplement si $f^n(x) = z, \forall x \in \mathsf{E}_n$ après s'être assuré que f admet un unique point fixe z. Remarquons qu'il suffit de tester si la n-ième itérée de x par f est égale à l'attracteur principal car la plus longue chaîne possible dans E_n est de longueur n.

```
def admet_attracteur_principal(t):
    n=len(t)
    if nb_points_fixes(t) != 1:
        return False
    attracteur=itere(t,0,n)
    for i in range(1,n):
        if itere(t,i,n)!= attracteur:
            return False
    return True
```

6 Utilisons la relation de récurrence proposée par l'énoncé après l'avoir simplement traduite dans une procédure récursive.

```
def temps_de_convergence(t,x):
    if t[x] == x:
        return 0
    else:
        return 1 + temps_de_convergence(t,t[x])
```

7 Afin de ne pas surcharger le code, commençons par écrire une procédure qui renvoie le maximum d'un tableau donné en argument.

```
def max_tableau(t):
    max = t[0]
    for i in range(len(t)):
        if t[i] > max:
            max = t[i]
    return max
```

Construisons en temps linéaire un tableau contenant l'ensemble des temps de convergence pour le tableau `t`. Il suffira alors de prendre le maximum de ce tableau. La construction d'un tableau `tc` des temps de convergence en temps linéaire se fait en utilisant la relation de récurrence $\texttt{tc}[\texttt{i}] = 1 + \texttt{tc}[\texttt{t}[\texttt{i}]]$, dans le cas où `i` n'est pas un point fixe. On va ainsi pouvoir remplir `tc`, bout de chaîne de convergence par bout de chaîne de convergence. Examinons le principe sur un exemple général :

- On commence par calculer `tc[0]`. Pour cela, si 0 est point fixe, on sait qu'il suffit de mettre la case `tc[0]` à 0.
- Sinon, on lance un appel récursif pour calculer `tc[t[0]]`. Cet appel récursif va en fait calculer récursivement tous les temps de convergence des entiers qui se situent sur la chaîne entre 0 et l'attracteur principal. Une fois ce travail terminé, il n'y a plus qu'à mettre la valeur de `tc[0]` à `tc[t[0]]+1`.
- Une fois cette première étape terminé, on passe au calcul de `tc[1]`. Pour cela, on commence par consulter la valeur de `tc[1]`. En effet, il est possible que 1 ait été rencontré sur la chaîne entre 0 et l'attracteur principal. Dans ce cas, il n'y a rien à faire. Sinon, on procède comme pour le calcul de `tc[0]`.
- On termine l'algorithme en appliquant le principe utilisé pour calculer `tc[0]` puis `tc[1]` à tous les éléments du tableau.

La fonction `temps_de_convergence` s'écrit donc en deux étapes.

```
def maj_tc(t,tc,i):
    if t[i]==i:
        tc[i]=0
    else:
        maj_tc(t,tc,t[i])
        tc[i]=tc[t[i]]+1

def temps_de_convergence(t):
    n=len(t); tc=[-1]*n
    for i in range(n):
        if tc[i]==-1:
            maj_tc(t,tc,i)
    return max_tableau(tc)
```

Recherche efficace de points fixes

8 Vérifions que `t[i]` $\leq$ `t[i+1]`, $\forall$ `i` $\in [\![0; n-2]\!]$, à l'aide d'une boucle `for` que l'on interrompt si jamais cette condition n'est pas respectée.

```
def est_croissante(t):
    for i in range(0,len(t)-1):
        if t[i]>t[i+1]:
            return False
    return True
```

9 Procédons par dichotomie sur le tableau en appliquant le théorème du point fixe admis dans l'énoncé sur la moitié de tableau qui le respecte : en particulier, on maintient un tableau `t[a..b]` dont les valeurs sont dans $[\![$`a`$;$`b`$]\!]$. À chaque étape on regarde si l'élément du milieu du tableau `t[a..b]` considéré est un point fixe. Si ce n'est pas le cas, suivant sa position par rapport à son image par `t` (c'est-à-dire par rapport à la droite $y = x$), on conserve la première ou seconde moitié du tableau.
La fonction f étant à valeurs dans $[\![a; b]\!]$, on a `t[a]` $\geq$ `a` et `t[b]` $\leq$ `b`. Ainsi, si l'on note $c = \lfloor (a+b)/2 \rfloor$ le milieu de l'intervalle $[\![a; b]\!]$, on peut distinguer trois cas :

- soit `t[c]` $=$ `c` et alors c est un point fixe
- soit `t[c]` $<$ `c` et alors `t[a..c]` est à valeurs dans $[\![a; c]\!]$
- soit `t[c]` $>$ `c` et alors `t[c..b]` est à valeurs dans $[\![c; b]\!]$

Dans les deux derniers cas, le théorème du point fixe nous permet de rechercher un point fixe dans l'intervalle d'indices correspondant.
Afin d'éviter de recopier une partie du tableau, écrivons une fonction `pt_fixe` qui prend en plus deux entiers `a` et `b` et qui trouve un point fixe sur `t[a..b]`.

```
def pt_fixe(t,a,b):
    c=(a+b)//2
    if t[c]==c:
        return c
    elif t[c]<c:
        return(pt_fixe(t,a,c))
    else:
        return(pt_fixe(t,c,b))
```

On appelle ensuite cette fonction sur l'ensemble du tableau `t` dans `point_fixe`.

```
def point_fixe(t):
    return pt_fixe(t,0,len(t)-1)
```

10 Dans un algorithme qui procède par dichotomie, la variable qui décroît strictement à chaque itération est la taille de la partie du tableau sur laquelle on travaille, c'est-à-dire `b` $-$ `a` : elle est divisée par deux à chaque étape. Cette taille est positive ou nulle. Lorsqu'elle s'annule on a `a` $=$ `b`. Comme on a maintenu la propriété `t[a..b]` est à valeurs dans $[\![$`a`$;$`b`$]\!]$, on a trouvé un point fixe et la procédure termine immédiatement. Ainsi,

La procédure termine en un nombre fini d'appels récursifs.

On peut aussi rajouter grâce à cette analyse que le nombre d'appels récursifs est borné par $\lceil \log_2(n) \rceil$.

Remarque 2

Afin d'expliciter un peu plus le raisonnement qui suit sans surcharger le corrigé, repartons de la définition du temps de calcul $T($`point_fixe`$, n)$: il s'agit d'un maximum sur tous les tableaux `t` de taille n du nombre d'instructions lors de l'exécution de `point_fixe` sur `t`.
Pour montrer une borne supérieure sur ce maximum, on doit considérer un tableau quelconque de taille n et montrer que le nombre d'instructions lors de l'exécution de `point_fixe` sur `t` est majoré par $\beta \log(n)$ avec une constante β indépendante du tableau et de n.
Au contraire, pour montrer une borne inférieure sur ce maximum, il suffit d'exhiber pour chaque taille n un tableau `t` particulier pour lequel le nombre d'instructions lors de l'exécution de `point_fixe` sur `t` est toujours minoré par $\alpha \log(n)$ avec une constante α indépendante du tableau et de n.

Notons β' le nombre maximal d'instructions que l'on peut réaliser dans le pire des cas au cours de l'exécution d'un appel récursif de `pt_fixe` dans `point_fixe`. Alors comme on exécute au plus $\lceil \log_2(n) \rceil$ appels, si on part d'un tableau `t` de taille n, il vient que le nombre d'instructions lors de l'exécution de `point_fixe` sur `t` est majoré par $\beta'(\log_2(n)+1)$ (car $\lceil \log_2(n) \rceil \leq \log_2(n)+1$) qui est majoré par $(\beta'+1)\log_2(n)$ pour n assez grand.
Examinons maintenant la borne inférieure : soit f_n la fonction de E_n dans E_n constante égale à 1. Alors, pour n de la forme 2^k, la procédure `point_fixe` sur le tableau représentant f_n nécessite k appels récursifs. En notant α le nombre minimal d'instructions que l'on peut réaliser dans le pire des cas au cours de l'exécution d'un appel de `pt_fixe`, on obtient donc que sur ce tableau, la procédure `point_fixe` exécute au moins $\alpha\lceil log2(n) \rceil$ instructions, ce qui est minoré par $\alpha \log_2(n)$. En notant $\beta = \beta' + 1$, on obtient

$$\alpha \log(n) \leq T(\texttt{point\_fixe}, n) \leq \beta \log(n).$$

Ainsi, La procédure `point_fixe` s'exécute en temps logarithmique.

11 Soit $x \in \mathsf{E}_n$ tel que $f(x) = x$. Par définition de m on a $m \preceq x$. En appliquant k fois f, qui est croissante au sens de $\preceq$, on obtient au final :

$$f^k(m) \preceq f^k(x) = x.$$

Ainsi, $f^k(m)$ est le plus petit point fixe au sens de $\preceq$.

12 D'après la question 11, le plus petit point fixe est de la forme $f^k(1)$ car 1 est le plus petit élément de E_n au sens de l'ordre de divisibilité. Soit alors x_i un point fixe de f : $f(x_i) = x_i$. Comme 1 divise tous les entiers on a $1 \mid x_i$ et par k applications de f, qui est croissante au sens de l'ordre de divisibilité, il vient $f^k(1) \mid f^k(x_i)$. Ainsi, $f^k(1)$ divise tous les points fixes de f. Par conséquent, $f^k(1)$ divise le pgcd d de l'ensemble des points fixes de f. Or $f^k(1)$ est l'un de ces points fixes, donc $d \mid f^k(1)$, d'où $f^k(1) = d$. Ainsi,

Le plus petit point fixe de f est le pgcd de l'ensemble des points fixes de f.

13 En partant de 1, appliquons f jusqu'à trouver un point fixe : il s'agit du plus petit d'après la question 11 et du pgcd des points fixes d'après la question 12. Ceci est réalisé grâce à une boucle `while`.

```
1 def pgcd_point_fixe(t):
2     i=1
3     while t[i] != i:
4         i=t[i]
5     return i
```

14 Comme 1 divise tous les entiers et par croissance de f au sens de l'ordre de divisibilité, on a $1 \mid f(1) \mid f^2(1) \mid \cdots \mid f^k(1)$. De plus,

$$\begin{aligned} a \mid b \text{ et } a \neq b &\Rightarrow \exists c \geq 2 \quad b = ca \\ &\Rightarrow b \geq 2a \end{aligned}$$

Avant chaque tour de boucle, on a `i` $\mid$ `t[i]` et `t[i]` $\neq$ `i`, on a ainsi `t[i]` $\geq$ `2i`. On multiplie au moins par deux la valeur de `i` à chaque étape. Comme $\mathsf{E}_n = [\![0; n-1]\!]$, il y a au plus $\lfloor \log_2(n) \rfloor$ étapes. Comme chaque tour de boucle s'exécute en un nombre fini d'instructions, la borne supérieure sur $T(\texttt{pgcd\_point\_fixe}, n)$ se prouve de manière identique à la question 10. Pour la borne inférieure, considérons la fonction f_n de E_n dans E_n telle que

$$f_n(x) = \begin{cases} 2x \text{ si } 2x < n \\ 0 \text{ sinon} \end{cases}$$

f_n est bien croissante au sens de l'ordre de divisibilité, pour tout $x, y \in \mathsf{E}_n$: si $x \mid y$ alors $2x \mid 2y$ et $x \mid 0$.
Pour n de la forme 2^k, la procédure `pgcd_point_fixe` sur le tableau représentant f_n nécessite k itérations. En reprenant le raisonnement de la question 10, il vient qu'il existe $\alpha > 0$ tel que sur ce tableau, la procédure `pgcd_point_fixe` exécute au moins $\alpha \log_2(n)$ instructions.

Ainsi, La procédure `pgcd_point_fixe` s'exécute en temps logarithmique.