Concours Polytechnique filières MP, PC : corrigé

Jean-Loup Carré

Informatique commune – 2013

Question 1.
```
def admet_point_fixe(t):
    for i in range(len(t)):
        if t[i]==i:
            return True
    return False
```

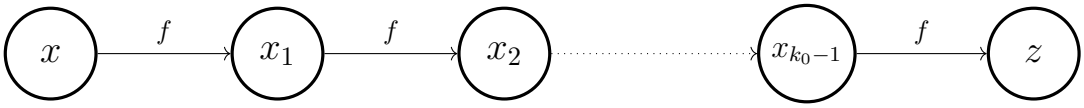
Question 2.
```
def nb_points_fixes(t):
    c = 0
    for i in range(len(t)):
        if t[i]==i:
            c += 1
    return c
```

Question 3.
```
def itere(t, x, k):
    for i in range(k):
        x = t[x]
    return x
```

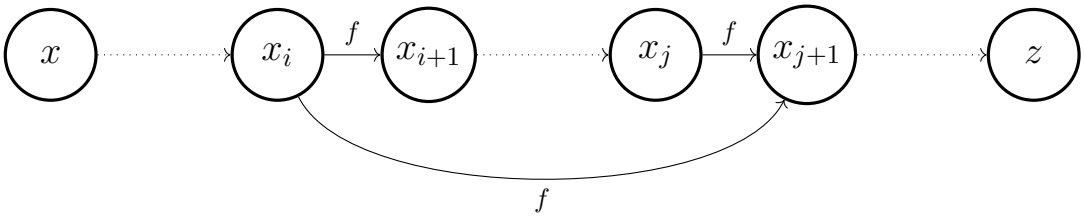
Question 4.
```
def nb_points_fixes_iteres(t, k):
    c = 0
    for i in range(len(t)):
        if itere(t, i, k)==i:
            c += 1
    return c
```

Question 5. Supposons que f ait un attracteur principal z. Soit $x \in E_n$, notons $x_k = f^k(x)$ et posons k_0 le plus petit des entiers k tels que $f^k(x) = z$.

Examinons le chemin menant de x à z :



Par l'absurde, supposons qu'il existe une répétition dans ce chemin, c'est-à-dire qu'il existe deux entiers i et j tels que $0 \leqslant i < j \leqslant k_0$ et $f^i(x) = f^j(x)$.



On remarque alors qu'il existe un chemin strictement plus court, qui court-circuite $x_{i+1}, \ldots, x_j$. Autrement dit, pour $k_1 = k_0 - (j - i) < k_0$, on a $f^{k_1}(x) = z$ ce qui contredit la minimalité de k_0.

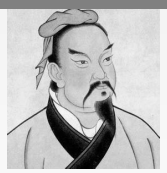
Ainsi, tous les éléments du chemin sont différents. Comme il y a n valeurs possibles, le chemin est de longueur au plus $n - 1$.

En conclusion, si f admet un attracteur principal, alors f^{n-1} est constante (la réciproque est triviale).

On en déduit l'algorithme demandé, qu'on traduit comme suit en Python :

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

Conseil stratégique



Sun Tsu

Il ne faut jamais hésiter à faire un dessin pour rendre le propos plus clair et surtout plus rapidement compréhensible par le lecteur.

Question 6.

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

Question 7. Pour résoudre le problème en temps linéaire, on va raisonner en plusieurs temps. Tout d'abord, considérons la solution naïve (qui est en temps quadratique) :

```
def temps_de_convergence_max_naif(t):
    M = 0 # Le temps de convergence vaut au moins zéro
    for x in range(len(t)):
        M = max(temps_de_convergence(t, x), M)
    return M
```

Examinons d'où vient la complexité en reprenant les notations de la question **5**.

Lors du calcul du temps de convergence de x, on va calculer, par récursivité, les temps de convergence de x, x_1, x_2, …
Lors du calcul du temps de x_1, on va calculer les temps de x_1, x_2, …
Lors du calcul du temps de x_2, on va calculer les temps de x_2, x_3, …
Pour éviter de refaire plusieurs fois le même calcul, nous allons modifier la fonction `temps_de_convergence` (appelons `tc` la fonction modifiée) pour qu'elle sauvegarde dans un tableau `R` les résultats qu'elle a déjà calculés.

Ce tableau `R` dépend de `t` (car il a la même longueur) et doit être extérieur à `tc` (pour ne pas être réinitialisé à chaque appel récursif). Cela nous amène à définir `tc` comme une sous-fonction de la fonction `temps_de_convergencence_max`.

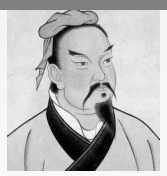
Finalement, le code suivant répond à la question :

```
def temps_de_convergence_max(t):
    R = [None] * len(t)
    def tc(x):
        if R[x] != None: # Si tc(x) n'a pas déjà été calculé
            if t[x] == x:
                R[x] = 0
            else :
                R[x] = 1 + tc(t[x])
        return R[x]
    M = 0
    for k in range(len(t)):
        M = max(tc(x), M)
    return M
```

Même code que pour la fonction naïve en remplaçant la fonction `temp_de_convergence` par `tc`.

Dans ce code, `R[x]` vaut `None` si `tc(x)` n'a jamais été calculé, sinon, il vaut le résultat déjà calculé de `tc(x)`.

Conseil stratégique


Sun Tsu

Nous utilisons ici une méthode classique, la *mémoïzation*, qui consiste à ajouter un *cache* (ici `R`) qui se souvient des résultats déjà calculés de la fonction récursive `tc`.

Lorsqu'on utilise une construction difficile à relire (comme ici une fonction à l'intérieur d'une autre fonction), il est important d'expliquer ce qu'on fait pour être compris du correcteur.

Question 8.
```
def est_croissante(t):
    for k in range(len(t)-1):
        if t[k] > t[k+1]:
            return False
    return True
```

Question 9. On va utiliser le principe de dichotomie.

```
def point_fixe(t):
    a = 0
    b = len(t)
    while a<b:
        c = (a+b)//2
        if t[c] == c:
            return c
        elif t[c] < c:
            b = c
        else :
            a = c+1
```

Cette fonction est correcte, car le `while` maintient l'invariant suivant : « $f(a) \geqslant a$ et $f(b-1) \leqslant b-1$ » et donc la restriction de f à $\{a; a+1; \dots b-1\}$ est croissante et à valeur dans $\{a; a+1; \dots b-1\}$ donc admet un point fixe. Ainsi, f admet toujours un point fixe entre a inclus et b exclus.

Si f n'est pas croissante, la fonction peut ne jamais exécuter le `return c`. Dans ce cas, elle renvoie `None`.

Question 10. Posons $i = b-a$. Nous avons trois cas dans le code (`if`, `if`, `else`). Dans le premier cas, la fonction s'arrête, dans le deuxième, i devient $\left\lfloor \frac{i}{2} \right\rfloor$, dans le troisième cas, i devient $\left\lceil \frac{i}{2} \right\rceil - 1$ (à cause du `c+1`).

Ainsi, soit la boucle s'arrête, soit i décroît strictement. De plus, i reste toujours positif (condition du `while`), donc la boucle termine.

On remarque de plus qu'à chaque étape i est au moins divisé par deux. Ainsi, au bout de k étapes, (en notant n le nombre d'élément de t), on a : $i \leqslant \frac{n}{2^k}$.

Cherchons k tel que $\frac{n}{2^k} < 1$. En résolvant on obtient $\log_2(n) < k$. Ainsi, en posant $k_0 = \lfloor \log_2(n) \rfloor + 1$ on a $\frac{n}{2^{k_0}} < 1$. Donc, au bout de k_0 étapes, si la boucle ne s'est pas terminé, alors $b = a$ (absurde, car ça implique que f n'a pas de point fixe), donc la boucle termine en moins de k_0 étapes, donc en temps logarithmique.

Question 11. Montrons par récurrence sur $p \in \mathbb{N}$ la propriété P_p plus générale suivante : « Pour tout $p \in \mathbb{N}$ et tout point fixe z de f, $f^p(m) \preceq z$ ».

Initialisation : comme m est un plus petit élément, $f^0(m) = m \preceq z$.

Hérédité : Soit z un point fixe de f, par hypothèse de récurrence $f^p(m) \preceq z$, par croissance de f on a $f(f^p(m)) \preceq f(z)$ donc $f^{p+1}(m) \preceq f(z) = z$.

On en déduit dans le cas particulier où $f^k(m)$ est lui-même un point fixe que c'est le plus petit point fixe.

Question 12. Comme 1 est minimal pour la divisibilité, $1 \mid f(1)$, donc, comme f est croissante (et donc aussi les f^k, qui sont des composées de fonctions croissantes), on a $\forall k \in \mathbb{N}, f^k(1) \mid f^{k+1}(1)$. On en déduit que la suite des $\left(f^k(1)\right)_{k\in\mathbb{N}}$ est croissante dans un ensemble fini, donc stationnaire. Appelons ℓ sa valeur finale. On a alors $f(\ell) = \ell$ par définition et donc ℓ est un point fixe. De la question précédente, on déduit que ℓ est le plus petit point fixe.

Soit d le pgcd de tous les points fixes. Comme ℓ divise tous les points fixes, ℓ divise leur pgcd, donc $\ell \mid d$. Or ℓ est un point fixe, et, par définition, d divise tous les points fixes, donc $d \mid \ell$ et par antisymétrie, $\ell = d$.

Remarque

Le dernier paragraphe est inutile si on sait que le pgcd est la borne inférieure pour l'ordre de divisibilité. On déduit du fait que ℓ est le minimum des points fixes le fait qu'il est aussi la borne inférieure de l'ensemble des points fixes.

Question 13. On utilise le fait, démontré à la question précédente, que la suite de $f^k(1)$ est stationnaire, et que sa valeur finale est le pgcd recherché.

```
def pgcd_points_fixes(t):
    x = 1
    while t[x] != x:
        x = t[x]
    return x
```

Question 14. Du fait que $f^k(1) \mid f^{k+1}(1)$ pour tout k, on déduit qu'à chaque passage dans le `while` x est soit multiplié par 2 ou plus, soit multiplié par zéro (et dans ce cas, le `while` s'arrête à la prochaine itération).

On en déduit qu'à la k-ème itération (sauf éventuellement la dernière), $n > x \geqslant 2^k$. Soit k_0 l'avant-dernière itération, on a alors $2^{k_0} < n$ et donc $k_0 < \log_2(n)$ d'où une complexité logarithmique.