Corrigé : points fixes de fonctions à domaine fini (X mp 2013)

Partie I. Recherche de point fixe, cas général

Question 1.

```
def admet_point_fixe(t):
    for x, fx in enumerate(t):
        if fx == x:
            return True
    return False
```

Rappelons que la fonction `enumerate` énumère les couples (indice, élément) d'un tableau.

Question 2.

```
def nb_points_fixes(t):
    s = 0
    for x, fx in enumerate(t):
        if fx == x:
            s += 1
    return s
```

Question 3.

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

Question 4.

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

Question 5. Considérons une fonction f admettant un attracteur principal z. Sachant que $\operatorname{card} \mathrm{E}_n = n$, pour tout $x \in \mathrm{E}_n$ il existe $i < j$ dans $[\![0, n]\!]$ tel que $f^i(x) = f^j(x)$. La suite $(f^k(x))_{k \geqslant i}$ est alors périodique de période $j - i$. Mais f admet un attracteur principal z donc nécessairement $f^i(x) = z$.
Ceci prouve que si f admet un attracteur principal z alors pour tout $x \in \mathrm{E}_n$, $f^{n-1}(x) = z$. Réciproquement, cette condition implique clairement que z est un attracteur principal, ce qui nous permet de choisir ce critère dans la fonction qui suit.

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

Question 6. L'énoncé suggère une version récursive de la fonction :

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

Question 7. La fonction de la question précédente a un coût linéaire $O(n)$; l'appliquer à tous les éléments du tableau conduit à un coût quadratique $O(n^2)$. Pour obtenir un coût linéaire, une solution consiste à la *mémoïser*, autrement dit à garder trace dans un tableau (ou dans un dictionnaire) des résultats intermédiaires. Ceci nous amène à redéfinir la fonction précédente :

```
def temps_de_convergence(t, tc, x):
    if tc[x] is None:
        if t[x] == x:
            tc[x] = 0
        else:
            tc[x] = 1 + temps_de_convergence(t, tc, t[x])
    return tc[x]
```

Le calcul du temps de convergence maximal ne pose alors plus de problème :

```
def temps_de_convergence_max(t):
    n = len(t)
    tc = [None for i in range(n)]
    m = 0
    for x in range(n):
        d = temps_de_convergence(t, tc, x)
        m = max(m, d)
    return m
```

Partie II. Recherche efficace des points fixes

Question 8. On observe qu'une fonction est croissante si et seulement si pour tout $x \in [\![0, n-2]\!]$, $f(x) \leqslant f(x+1)$.

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

Question 9. Un coût logarithmique suggère un algorithme de recherche dichotomique : on a $0 \leqslant f(0)$ et $f(n-1) \leqslant n-1$. On considère $k = \lfloor n/2 \rfloor$.

- Si $f(k) = k$, la recherche est terminée ;
- Si $f(k) < k$, la recherche se poursuit dans l'intervalle $[\![0, k-1]\!]$;
- Si $k < f(k)$, la recherche se poursuit dans l'intervalle $[\![k+1, n-1]\!]$.

```
def point_fixe(t):
    i, j = 0, len(t)
    while i + 1 < j:
        k = (i + j) // 2
        if t[k] == k:
            return k
        elif t[k] < k:
            j = k
        else:
            i = k + 1
    return i
```

Question 10. La terminaison de cet algorithme est assurée par la décroissance stricte de la quantité `j-i` ; sa validité par le maintient de l'invariant : $i \leqslant f(i)$ et $f(j-1) \leqslant j-1$. En effet, si la boucle conditionnelle se termine sans qu'un point fixe ait été trouvé, on a $j = i+1$ donc l'invariant procure les relations $i \leqslant f(i)$ et $f(i) \leqslant i$ qui prouvent que $f(i) = i$.
Notons u_k la valeur de `j-i` à l'entrée de la $(k+1)^{\text{e}}$ boucle. On dispose des relations $u_0 = n$ et $u_{k+1} \leqslant \dfrac{u_k}{2}$ donc $u_k \leqslant \dfrac{n}{2^k}$. Si on pose $p = \lceil \log n \rceil$ on est assuré que cet algorithme effectue au plus p fois la boucle conditionnelle, ce qui assure un coût logarithmique $\mathrm{O}(\log n)$.

Question 11. Soit x un point fixe de f. Puisque m est un plus petit élément de E on a $m \preccurlyeq x$. Puisque f est croissante il en est de même de f^k (ceci se prouve sans peine par récurrence) donc $f^k(m) \preccurlyeq f^k(x) = x$. $f^k(m)$ est le plus petit des points fixes.

Question 12. Soit d le plus petit des points fixes de f. D'après la question précédente, pour tout $k \in [\![1, m]\!]$, d divise x_k donc d divise $\mathrm{pgcd}(x_1, x_2, \ldots, x_m)$. Par ailleurs, il existe $i \in [\![1, m]\!]$ tel que $d = x_i$ donc $\mathrm{pgcd}(x_1, x_2, \ldots, x_m)$ divise d. Il résulte de ceci que $d = \mathrm{pgcd}(x_1, x_2, \ldots, x_m)$.

Question 13. 1 est un plus petit élément de E_n pour la relation de divisibilité. D'après la question 11 il existe $k \in \mathbb{N}$ tel que $f^k(1)$ soit le plus petit des points fixes de f dans E_n. D'après la question 12 c'est aussi le pgcd des points fixes de f.

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

Question 14. On a $1 \preccurlyeq f(1)$ et f est croissante, donc $f^k(1) \preccurlyeq f^{k+1}(1)$. Ceci signifie que $f^k(1)$ divise $f^{k+1}(1)$.
Ainsi, tant que le plus petit des points fixes n'est pas atteint, on a $f^{k+1}(1) \geqslant 2f^k(1)$ ou $f^{k+1}(1) = 0$.
Notons que s'il existe k tel que $f^{k+1}(1) = 0$ alors $f^{k+2}(1) = 0$ et le plus petit des points fixes est trouvé. Ainsi, tant que le plus petit des points fixes n'est pas trouvé on a $f^k(1) \geqslant 2^k$, ce qui montre que le processus de recherche se termine à un rang $k \leqslant \log n$. Cet algorithme est donc de coût logarithmique.