Corrigé du Devoir Maison 1 d'Informatique

Mardi 3 janvier 2017

1. La première variante a pour principe de remplir, en parcourant les indices x de 0 à $n-1$ et en évaluant $t(x)$, de remplir peu à peu une liste de booléen `antecendent_trouve` tel que `antecedent_trouve[k]` vaut `True` si, et seulement si, parmi les test effectués, on a trouvé un x tel que `t[x] == k`.

On utilise le fait qu'une application $t : E_n \to E_n$ injective est une permutation (propriétés des applications entre ensembles finis).

```
def estPermutation(t):
    n = len(t)
    antecedent_trouve = [False]*n
    for x in range(n):
        if t[x] not in range(n):
            return False
        elif antecedent_trouve[t[x]]:
            return False
        else:
            antecedent_trouve[t[x]] = True
    return True
```

La deuxième variante propose de remplir, sur le même principe, la liste `distrib_t` telle que `distrib_t[y]` désigne le nombre d'antécédents de y par t. Il est alors clair, indépendamment de propriétés spécifiques aux applications entre ensembles finis, que t est une permutation si, et seulement si, `distrib_t` est une liste ne comportant que des 1.

```
def estPermutation(t):
    n = len(t)
    distrib_t = [0]*n
    for x in range(n):
        if t[x] in range(n):
            distrib_t[t[x]] += 1
    return distrib_t == [1] * n
```

2. Voici diverses variantes, selon un parcours par valeurs ou par indices, et selon une construction en remplissant une liste initialisée à une valeurs n'ayant pas de sens (ici -1) par une boucle ou, ou bien une construction par compréhension.

Boucle `for` et parcours par les indices.

```
def composer(t, u):
    n = len(t)
    tou = [-1] * n
    for x in range(n):
        tou[x] = t[u[x]]
    return l
```

Construction par compréhension et parcours sur les indices.

```
def composer(t, u):
    n = len(t)
    return [t[u[x]] for x in range(n)]
```

Boucle `for` et et parcours sur les valeurs de u (mais ceci nécessite une incrémentation manuelle de l'indice x).

```
def composer(t, u):
    n = len(t)
    tou = [-1] * n
    x = 0
    for y in u:
        tou[x] = t[y]
        x += 1
    return tou
```

Construction par compréhension et parcours sur les valeurs de u.

```
def composer(t, u):
    return [t[y] for y in u]
```

Boucle `for` et parcours sur les couples (indice, valeurs) de u (fonction `enumerate`) Construction par compréhension et parcours sur les valeurs de u.

```
def composer(t, u):
    n = len(t)
    tou = [-1] * n
    for x, y in enumerate(u):
        tou[x] = t[y]
    return tou
```

3. Pour cette question, il n'y a pas de constructions immédiate par compréhension. Voici une première variante avec un parcours des indices de t.

```
def inverser(t):
    n = len(t)
    inv_t = [-1] * n
    for x in range(n):
        inv_t[t[x]] = x
    return inv_t
```

et une deuxième avec un parcours des couples (indices, valeurs) de t.

```
def inverser(t):
    n = len(t)
    inv_t = [-1] * n
    for x, y in enumerate(t):
        inv_t[y] = x
    return inv_t
```

4. L'identité Id_{E_n} est l'unique permutation de E_n d'ordre 1. Une exemple de permutation de E_n d'ordre n est l'application $\tau : x \mapsto x+1 \mod n$ (reste de la division euclidienne de $x+1$ par n). On peut en effet montrer par récurrence que, pour tout k, et tout x, $\tau^k(x) = x+k \mod n$. De fait, les k pour lesquels $\tau^k = \mathrm{Id}_{E_n}$ sont exactement les multiples de n, et le plus petit d'entre eux est bien n.

5. Le principe est simplement de calculer les itérés successifs (pour la composition) de t, puis de s'arrêter lorsque l'on tombe sur l'identité (en prenant évidemment soin de compter le nombre d'itérations, vu qu'il s'agit du résultat attendu).

```
def ordre(t):
    n = len(t)
    identite = list(range(n))
    k = 1
```

```
    tk = t[:]
    while tk != identite:
        k += 1
        tk = composer(t, tk)
    return k
```

L'invariant de boucle du programme est « la variable `tk` représente t^k ».

Le fait que la boucle `while` n'est pas infinie est dû au fait que l'ordre o d'une permutation de E_n existe dans $\mathbb{N}$ (et alors un variant de boucle adéquat est $o - k$).

Un moyen de prouver (mathématiquement) que l'ordre d'une permutation est fini tient en l'utilisation judicieuse des résultats suivants :

— L'application $k \mapsto t^k$ est un morphisme de groupes de $(\mathbb{Z}, +)$ dans le groupe $(\mathfrak{S}(E_n), \circ)$ des permutations de E_n (traduction savante de l'identité : $t^{k_1+k_2} = t^{k_1} \circ t^{k_2}$ pour tout k_1, $k_2 \in \mathbb{Z}$).

— Le noyau d'un morphisme de groupes de $(\mathbb{Z}, +)$ vers un groupe (quelconque) est un sous-groupe de $(\mathbb{Z}, +)$.

— Les sous-groupes de $(\mathbb{Z}, +)$ sont les parties de $\mathbb{Z}$ de la forme $a\mathbb{Z} = \{na \ ; \ n \in \mathbb{Z}\}$ pour $a \in \mathbb{N}$ fixé.

— Un morphisme de groupes est injectif si, et seulement si son noyau est réduit au singleton élément neutre (ici $\{0\} = 0\mathbb{Z}$).

Attention au problème (spécifique au langage Python lié aux copies de listes) : il faut veiller à ce que le résultat final ne modifie pas la liste t de départ, donc en faire une copie. Le moyen le plus concis pour le faire est d'utiliser la technique de slicing : `t[:]` est une copie (superficielle) de t, et non plus t elle-même. On aurait aussi pu écrire bien entendu `tk = [y for y in t]` ou encore |`tk = [t[x] for x in range(len(t))]`, au lieu de `tk = t[:]`.

6. C'est une version simplifiée de la fonction `ordre` de la question précédente. Les arguments employés s'adaptent point par points.

```
def periode(t, i):
    k = 1
    tk_i = t[i]
    while tk_i != i:
        tk_i = t[tk_i]
        k += 1
    return k
```

L'invariant de boucle est « `tk_i` prend la valeur $t^k(i)$ ».

7. Le premier programme proposé repose sur le principe suivant : si $(x_p)_{p\in\mathbb{N}}$ désigne la suite des itérés successifs de i par p (c'est-à-dire si $x_0 = i$ et $x_{p+1} = t(x_p)$) est o-périodique, ou o représente l'ordre de i. Par conséquent, en calculant les itérés successifs de i :

— ou bien l'on tombe d'abord sur i (avant d'avoir rencontré j) comme résultat. Dans ce cas, il est certain que j n'est pas dans l'orbite de i.

— ou bien l'on tombe sur j avant de rencontrer i (et évidemment, j est dans l'orbite de i

```
def estDansOrbite(t, i, j):
    k = 1
    tk_i = t[i]
    while tk_i != j and tk_i != i:
        tk_i = t[tk_i]
        k += 1
    return tk_i == j
```

Le second programme calcule l'orbite de i, pour vérifier si j en fait partie, ou non, en conclusion.

```
def estDansOrbite(t, i, j):
    orbite_de_i = [i]
    x = t[i]
    while x != i:
        orbite_de_i.append(x)
        x = t[x]
    return j in orbite_de_i
```

8. Une permutation est une transposition si, et seulement si, le nombre de ses points qui ne sont pas fixes est au nombre de deux : on peut donc compter le nombre de ses points fixes pour conclure, et c'est ce que propose le programme suivant.

```
def estDansOrbite(t, i, j):
    orbite_de_i = [i]
    x = t[i]
    while x != i:
        orbite_de_i.append(x)
        x = t[x]
    return j in orbite_de_i
```

9. Une permutation est un cycle si, et seulement si :

— l'ensemble de ses points fixes est l'orbite de l'un d'entre eux

— ceci équivaut (dans la mesure où toute orbite non réduite à un élément est un ensemble de deux points non fixes, et où, si deux ensembles A et B sont finis avec même nombre d'éléments et que $A \subset B$, alors $A = B$) à ce qu'il y ait autant de points non fixes que d'éléments d'une orbite d'un point non fixe x.

Le premier programme se sert littéralement du premier point :

```
def estCycle(t):
    n = len(t)
    x = 0
    while x < n and t[x] == x:
        x += 1
    if x == n:
        t_est_cycle = False
    else:
        t_est_cycle = True
        for y in range(x, n):
            if t[y] != y and not estDansOrbite(t, x, y):
                t_est_cycle = False
                break
    return t_est_cycle
```

Le second se sert littéralement du second point énoncé plus haut :

```
def estCycle(t):
    n = len(t)
    nb_points_non_fixes = 0
    point_non_fixe_trouve = False
    x = -1
    for y in range(n):
        if t[y] != x:
            np_points_non_fixes +=1
            if not point_non_fixe_trouve:
                x = k
```

```
                point_non_fixe_trouve = True
    return (point_non_fixe_trouve
            and nb_points_non_fixes == periode(t, x) )
```

10. Cette question est vraisemblablement la plus difficile du sujet.

Exemple à ne pas faire : Voici un exemple de programme qui n'aurait rapporté aucun point, dans la mesure où la complexité totale est très loin d'être linéaire en n.

```
def periodes(t):
    n = len(t)
    return [periode(t, i) for i in range(n)]
```

Voici comment on va s'y prendre :

— On initialise le tableau p des orbites à 0

— On crée un second tableau de booléens `orbite_trouvee`, initialisé à `False`, destiné à être tel que `orbite_trouvee[k]` vaut `True` si, et seulement si on a réussi à mettre k dans l'orbite d'un élément déjà parcouru.

— On parcoure le tableau t et, tombé sur un élément i sans orbite déjà trouvée, on crée l'orbite de i par une naïve boucle while (itérés successifs de i par t).

— Une fois l'orbite créée, les périodes des éléments de cette orbite sont toutes égales à la longueur de l'orbite, ce qui permet d'affecter consécutivement les cases de p correspondantes

Il n'est pas si évident que la complexité est linéaire, dans la mesure où il y a des boucles `while` et `for` imbriquées dans une boucle while. Toutefois, chaque entier j n'appartenant qu'à une seule orbite (qui, elle, est l'orbite de chacun de ses éléments), il n'a l'occasion de passer au travers des boucles `while j!= i` et `for j in orbite_de_i` qu'une seule fois.

```
def periodes(t):
    n = len(t)
    orbite_trouvee = [False] * n
    p = [0] * n
    for i in range(n):
        if not orbite_trouvee[i]:
            orbite_de_i =[i]
            j = t[i]
            while j != i:
              orbite_de_i.append(j)
              j = t[j]
            m = len(orbite_de_i)
            for j in orbite_de_i:
                orbite_trouvee[j] = True
                p[j] = m
    return p
```

11. Dans le programme suivant, la boucle en i a pour invariant « `t_i_j` prend pour valeur $t^i(j)$ », la boucle en j a pour invariant : « `t_k` a pour valeurs aux indices $x < i$ $t^k(x)$ ».

```
def itererEfficace(t, k):
    n = len(t)
    p = periodes(t)
    t_k = [0] * n
    for j in range(n):
        r = k % p[j]
        t_i_j = j
        for i in range(r):
            t_i_j = t[t_i_j]
        t_k[j] = t_i_j
    return t_k
```

12. La permutation $(0,1)(2,3,4)$ qui échange 0 et 1, et permute circulairement 2, 3, et 4, est d'ordre 6 sur E_5.

13. C'est du cours.

Le variant de boucle est r (la suite des valeurs prises par r est une suite strictement décroissante d'entiers naturels).

L'invariant de boucle est : PGCD(u, v) = PGCD(a, b). Au dernier tour de boucle, r prend une dernière fois une valeur non nulle à l'entrée, et $r = 0$ à la fin du tour, se sorte que v divise u, et donc PGCD(u, v) = v.

```python
def pgcd(a, b):
    r = a % b
    u = a
    v = b
    while r != 0:
        u = v
        v = r
        r = u % v
    return v
```

14. RAS.

```python
def ppcm(a, b):
    return (a * b) // pgcd(a, b)
```

15. On utilise, d'une part l'associativité du PPCM, d'autre part, le fait que si a divise x, alors PPCM(x, a) = x (ce qui rend inutile l'appel de la fonction `ppcm`).

```python
def ordreEfficace(t):
    ordre = 1
    n = len(t)
    for x in range(n)
        periode_x = periode(t, x)
        if ordre % periode_x != 0:
            ordre = ppcm(ordre, periode_x)
    return ordre
```