Épreuve d'informatique de l'X - 2011 - MP/PC

Pour être plus en accord en Python, j'ai travaillé avec des tableaux indexés de 0 à $n-1$ et travaillé dans les programmes avec les permutations sur l'ensemble $\{0, 1, \ldots, n-1\}$.

```
1  taille = len
2
3  def allouer(n) :
4      t = [0 for k in range(n)]
5      return(t)
6
7  def reste(a,b) :
8      return(a % b)
```

Ordre d'une permutation

Question 1. Pour vérifier qu'une tableau représente bien une permutation, on utilise un tableau `test` qui permet de signaler les entiers déjà rencontrés. Dans la boucle des lignes 6 à 11, on vérifie que j le i^{eme} élément du tableau est compris entre 0 et $n-1$ (où n est la taille du tableau `t`) et que j n'était pas présent dans la partie du tableau étudiée. Si l'une des conditions n'est pas vérifiée, on quitte le programme en retournant la valeur `False` (ligne 9) sinon on marque dans le tableau `test` la présence de j (ligne 11). Si on arrive à la ligne 12, c'est que les n éléments du tableau `t` sont distincts et compris entre 1 et n, le tableau t représente bien une permutation.

```
1  def estPermutation(t) :
2      n = taille(t)
3      test = allouer(n)
4      for i in range(n) :
5          test.append(False)
6      for i in range(n) :
7          j = t[i]
8          if j<0 or j>n-1 or test[j] :
9              return(False)
10         else :
11             test[j] = True
12     return(True)
```


Question 2. Pour tout entier i de $[\![0, n-1]\!]$, la valeur $(t \circ u)(i)$ est représentée par l'entier `t[u[i]]`. Dans la boucle des lignes 4 à 5, on remplit le tableau des résultats en conséquence.

```
1  def composer(t, u) :
2      n = taille(t)
3      res = allouer(n)
4      for i in range(n) :
5          res[i] = t[u[i]]
6      return res
```

Question 3 : Si $t(i) = j$ alors $t^{-1}(j) = i$. On applique cette formule à la ligne 5 pour générer le tableau représentant t^{-1} à partir du tableau représentant t.

```
1  def inverser(t) :
2      n = taille(t)
3      res = allouer(n)
4      for i in range(n) :
5          res[t[i]] = i
6      return res
```

Question 4 : La permutation $[1, 2, \ldots, n]$ est une permutation d'ordre 1 et la permutation $[2, 3, \ldots, n, 1]$ est une permutation d'ordre n.

Question 5 : On commence par écrire une fonction `EstIdentite` qui teste si une permutation est égale à l'identité. Elle retourne le booléen `True` si c'est le cas et le booléen `False` dans le cas contraire.

```
1  def EstIdentite(t) :
2      n = taille(t)
3      for i in range(n) :
4          if t[i] !=i :
5              return(False)
6      return True
```

Pour calculer l'ordre d'une permutation, on calcule successivement les t^k jusqu'à ce que l'on obtienne l'identité. Le tableau `tk` représente la permutation t^k. On quitte la boucle conditionnelle des lignes 7 à 9 dès que $t^k = Id_{[\![0,n-1]\!]}$. La variable de comptage `compt` contient la valeur du k correspondant.

```
1  def ordre(t) :
2      compt = 1
3      n = taille(t)
4      tk = allouer(n)
5      for i in range(n) :
6          tk[i] = t[i]
7      while not(EstIdentite(tk)) :
8          compt += 1
9          tk = composer(t,tk)
10     return compt
```

II. Manipuler les permutations

Question 6 : À chaque test dans la boucle conditionnelle des lignes 4 à 6, la variable `j` contient $t^k(i)$. On quitte cette boucle pour le premier entier k tel que $t^k(i) = i$, c'est-à-dire quand k correspond à la période de i pour la permutation t.

```
1  def periode(t,i) :
2      k = 1
3      j=t[i]
4      while j != i :
5          k += 1
6          j = t[j]
7      return k
```

Question 7 : Si p est la période de i alors l'orbite de i est $\{i, t(i), \dots, t^{p-1}(i)\}$. Pour savoir si l'entier j est dans l'orbite de t, on compare j avec les $t^\ell(i)$ ($0 \leqslant \ell < p$ où p est l'ordre de i). On quitte le programme si j coïncide avec l'un des points de l'orbite (ligne 4) et on retourne le booléen `True`. Si j n'est pas dans l'orbite de i, on exécute la boucle sans quitter le programme, celui-ci va donc retourner le booléen `False` (ligne 7).

```
1  def estDansOrbite(t, i, j) :
2      for l in range(periode(t,i)) :
3          if i == j :
4              return True
5          else :
6              i=t[i]
7      return(False)
```

Question 8 : Pour tester si une permutation est une transposition, on compte le nombre d'entiers qui ne sont pas invariants par la permutation `t`. Si ce nombre est égal à 2, c'est que `t` est une transposition et sinon ce n'est pas le cas.

```
1  def estTransposition(t) :
2      n = taille(t)
3      compt = 0
4      for i in range(n) :
5          if t[i] != i :
6              compt += 1
7      return(compt==2)
```

Question 9 : Plus généralement, pour tester si une permutation t est un cycle, il suffit de vérifier que le nombre d'éléments qui ne sont pas invariants par t est égal à l'ordre de la permutation t. C'est le principe du programme `estCycle` :

```
1  def estCycle(t) :
2      n = taille(t)
3      compt = 0
4      for i in range(n) :
5          if t[i] != i :
6              compt += 1
7      return(compt == ordre(t))
```

Question 10 : On commence par créer un tableau `p` des périodes que l'on initialise avec des 0. Puis, pour chaque entier i, si la période de i n'a pas été encore attribuée (donc `p[i]=0`) alors, on calcule la période p_i de l'élément i. Les éléments de l'orbite de i ayant la même période que i, on met la valeur p_i dans les cases du tableau `p` correspondant aux éléments de l'orbite de i. Le tableau des périodes est parcouru deux fois ainsi que celui représentant la permutation t. La complexité est bien linéaire.

```
1  def periodes(t) :
2      n = taille(t)
3      p = allouer(n)
4      for i in range(n) :
5          p[i] = 0
6      for i in range(n) :
7          if p[i] == 0 :
8              p_i = periode(t,i)
9              k = i;
10             for j in range(p_i) :
11                 p[k] = p_i
12                 k = t[k]
13     return(p)
```

Question 11 : Si p est la période de l'élément i et k' le reste de la division euclidienne de k par p, on a $t^k(i) = t^{k'}(i)$. Ce résultat est exploité dans le programme `itererEfficace` ci-dessous. On commence par calculer le tableau des périodes (ligne 4), puis pour chaque élément i, on calcule $t^{k'}(i)$ grâce à la boucle des lignes 6 à 9. Le résultat est stocké dans le tableau `res` (ligne 10).

```
1  def itererEfficace(t, k) :
2      n = taille(t)
3      res = allouer(n)
4      tab_periodes = periodes(t)
5      for i in range(n) :
6          l = i
7          for j in range(reste(k,tab_periodes[i])) :
8              l = t[l]
9          res[i] = l
10     return(res)
```

Question 12 : La permutation $[5, 4, 2, 3, 1]$ est d'ordre 6 et de taille 5.

Question 13 : On écrit une fonction `pgcd` qui repose sur l'algorithme d'Euclide.

```
1  def pgcd(a, b) :
2      while b != 0 :
3          tmp = b
4          b = reste(a,b)
5          a = tmp
6      return(a)
```

Question 14 : La fonction `ppcm` repose sur l'identité $\text{ppcm}(a,b) = \dfrac{a.b}{\text{pgcd}(a,b)}$.

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

Question 15 : On calcule le plus petit commun multiple des périodes des éléments. Pour limiter le nombre d'appels à la fonction `ppcm`, on utilise un tableau `marquage` qui permet de mémoriser les k qui ont déjà été pris en compte dans le calcul du ppcm.

```
1  def ordreEfficace(t) :
2      n = taille(t)
3      periodes_t = periodes(t)
4      marquage = allouer(n)
5      res = 1
6      for i in range(n) :
7          marquage[i] = True
8      for i in range(n) :
9          k = periodes_t[i]
10         if marquage[k] :
11             marquage[k] = False
12             res = ppcm(res,k)
13     return(res)
```