X Informatique PC 2011

Le sujet d'informatique de l'école polytechnique a du fréquemment d'adapter faute de l'existence d'un véritable enseignement d'un langage de programmation en CPGE avant la réforme. C'est la raison pour laquelle dans ce sujet les fonctions `allouer` et `taille` sont données. Par ailleurs, le fait que les indices des tableaux commencent à 1 provient clairement de la contrainte de coller à Maple, puisque c'est quasiment le seul outil informatique qui adopte cette convention pour ses tableaux !

Dans ce corrigé, on considère que les tableaux sont indexés à partir de 0, et qu'ils représentent des permutations de $[\![0; n-1]\!]$ où n est la taille du tableau. En python, on peut définir `allouer` de la façon suivante :

```
def allouer(n):
    return [0]*n
```

Pour la fonction `taille`, il suffit d'utiliser la commande `len` à la place.

1 Ordre d'une permutation

1 Si `t` est une permutation, alors par définition (c'est une bijection) on y trouve une et une seule fois chaque entier de $[\![1; n]\!]$. Pour tout indice de t, effectuons un test en deux parties :

- une première où l'on regarde si l'entier appartient bien à E_n,
- une seconde où l'on vérifie que l'entier n'est pas déjà apparu.

Dans le code, cela représente une boucle `for` dans laquelle les deux tests sont effectués. Pour vérifier que chaque entier apparaît une seule fois, utilisons un tableau de valeurs booléennes `present` de la même taille que t et initialisé à `False`. Chaque élément `present[i]` permet de savoir si l'entier `i` a déjà été rencontré. Si l'on rencontre `i` alors que `present[i]` vaut `True`, alors `t` n'est pas une permutation et le code renvoie `False` ; sinon `present[t[i]]` est mis à jour à `True`. Si on sort de la boucle `for` sans retourner `False`, alors le tableau t est constitué de n entiers deux à deux distincts appartenant à E_n, de cardinal n. Ainsi, t représente bien une permutation de E_n et le code renvoie `True`.

```
def estPermutation(t):
    n=len(t); present=[False]*n
    for i in range(n):
        if t[i]<0 or t[i]>=n or present[t[i]]:
            return False
        present[t[i]]=True
    return True
```

C'est ici la fonction `return` qui permet d'arrêter la procédure et de renvoyer la valeur `False`. L'exécution du programme s'interromp donc aussitôt dès qu'on constate une incohérence avec la représentation d'une permutation.

L'énoncé simplifie grandement la suite en indiquant que les arguments satisfont les contraintes imposées. Ainsi, il est inutile d'alourdir le code avec quantité de tests vérifiant par exemple que les tableaux donnés en arguments représentent bien des permutations ou qu'ils ont la même taille lorsqu'on va vouloir les comparer ou les composer.

2 Appliquons la formule de l'énoncé mot pour mot : il s'agit de calculer $t[u[i]]$ pour chaque i. On réalise cela grâce à une boucle `for` qui permet de parcourir tous les entiers de E_n. Les résultats sont stockés dans un tableau `composee` qui est retourné à la fin de la boucle.

```
def composer(t,u):
    n=len(t); s=allouer(n)
    for i in range(n):
        s[i]=t[u[i]]
    return s
```

3 Utilisons le fait que t est une bijection. Par définition, $t^{-1}[t[i]] = i$ pour tout entier $i \in E_n$. Autrement dit, si la case d'indice i de t contient la valeur $t[i]$, alors la case d'indice $t[i]$ du tableau à renvoyer doit contenir la valeur i. En utilisant une boucle `for`, on remplit à l'aide de cette remarque un tableau `inverse` représentant t^{-1}.

```
def inverse(t):
    n=len(t); u=allouer(n)
    for i in range(n):
        u[t[i]]=i
    return u
```

4 Cherchons deux permutations s et t telles que :

$$\forall i \quad s^1[i] = i \qquad \text{et} \qquad \forall i \quad t^n[i] = i$$

La première est triviale, il s'agit de l'identité elle-même (c'est d'ailleurs la seule permutation d'ordre 1). Pour la seconde, la plus simple est la permutation circulaire, c'est-à-dire la permutation vérifiant

$$t[0] = 1 \qquad t[1] = 2 \qquad \cdots \qquad t[n-2] = t[n-1] \qquad \text{et} \qquad t[n] = 1$$

Lorsque l'on compose k fois cette application par elle-même, on obtient l'application t telle que

$$t^k[0] = k \quad t^k[1] = [k+1] \qquad \cdots \qquad t^k[n-1-k] = n-1 \qquad \text{et} \qquad t^k[n-k] = 0 \qquad \cdots \qquad t^k[n-1] = k-1$$

En particulier, n est le plus petit entier k tel que t^k soit égal à l'identité.

Au final, L'identité est d'ordre 1 et la permutation circulaire $[1, 2, ..., n-2, 0]$ est d'ordre n.

> La théorie des groupes permet de montrer que l'ordre d'une permutation est toujours un diviseur de $n!$. Par conséquent, l'ordre d'une permutation peut être potentiellement largement supérieur à n.

5 Afin de faciliter l'écriture de la fonction `ordre`, commençons par écrire une fonction `estIdentite` qui prend en argument une permutation et renvoie `True` s'il s'agit de l'identité, `False` sinon. Pour faire ceci, une boucle `for` parcourt l'ensemble des indices du tableau et vérifie à l'aide d'une instruction `if` s'ils ont les bonnes valeurs, c'est-à-dire si $t[i] = i$ pour tout i.

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

A noter que python permet de le faire en une ligne avec le test `t=[i for i in range(n)]` mais par défaut, nous donnons la rédaction purement naïve (qui fait en fait la même chose).
Écrivons maintenant la fonction `ordre` à l'aide de notre fonction auxiliaire. Il s'agit de chercher le plus petit $k \in \mathbb{N}^*$ tel que t^k est égale à l'identité. Ne sachant pas combien d'itérations vont être nécessaires, utilisons une boucle `while`. La boucle devra s'arrêter lorsque $t^k[i] = i$ pour tout i, c'est-à-dire que `estIdentite(t)` vaut `True`. La fonction `composer` écrite précédemment permet de calculer les puissances successives de t. Elles sont successivement stockées dans un tableau `puissanceT` que l'on prend soin d'initialiser à t avec une boucle `for`. Il ne faut pas oublier non plus d'incrémenter le compteur `k` à chaque itération de la boucle `while`.

```
def ordre(t):
    k=1; u=copy.deepcopy(t)
    while not(estIdentite(u)):
        u=composer(u,t)
        k+=1
    return k
```

> Le calcul d'une composition se faisant en $O(n)$ opérations élémentaires, la fonction `ordre` est de complexité $O(nk)$ où k est l'ordre de la permutation t. Comme mentionné précédemment, cet ordre peut être relativement grand, ce qui rend la fonction de complexité particulièrement mauvaise.

2 Manipuler les permutations

6 Calculons ici uniquement $t^k[i]$ à chaque itération. Cette valeur est stockée dans une variable `iteree` en utilisant la relation de récurrence $t^{k+1}[i] = t[t^k[i]]$. La boucle `while` est stoppée quand on retombe sur la valeur i initiale.

```
def periode(t,i):
    c=1; p=t[i]
    while p!=i:
        p=t[p]
        c+=1
    return c
```

> On aurait pu utiliser la question précédente pour répondre à celle-ci et calculer les puissances t^k de toute la permutation, en s'arrêtant cette lorsque $t^k[i] = i$. C'est d'une part inutile : on n'a besoin que des images de i par les itérés de t, et pas du reste des images. C'est surtout nuisible car l'objectif de la suite est justement d'améliorer la complexité désastreuse de la version précédente de la fonction `ordre`.

7 Pensons quelque peu à la fonction écrite à la question précédente pour construire celle-ci. Parcourons l'orbite de i en calculant successivement les itérées $(t^k[i])_{k\in\mathbb{N}}$. Dès que $t^k[i] = j$, on retourne `True`, sinon on retourne `False` après avoir parcouru l'ensemble de l'orbite, c'est-à-dire une fois que l'on retombe sur la valeur i.

```
def estDansOrbite(t,i,j):
    if i==j:
        return True
    p=t[i]
    while p!=i:
        if p==j:
            return True
        p=t[p]
    return False
```

8 D'après la définition, une transposition est une permutation qui diffère de l'identité en exactement deux indices. La fonction `estTransposition` parcourt l'ensemble des indices de t et incrémente un compteur `compte` à chaque fois que l'on rencontre un indice tel que $t[i] \neq i$. Il suffit ensuite de vérifier que ce compteur vaut bien 2.

```
def estTransposition(t):
    c=0; n=len(t)
    for i in range(n):
        if t[i]!=i:
            c+=1
    return (c==2)
```

9 On souhaite ici vérifier que tous les indices de t pour lesquels $t[i] \neq i$ sont bien dans la même orbite. On va donc parcourir les indices de t et repérer le premier indice p tel que $t[i] \neq i$. Dès que cet indice est repéré, on crée un tableau de booléens dans lequel on met à True les indices correspondant à des éléments de l'orbite du cycle de p. Une fois ce travail fait, la dernière étape consiste à vérifier que tous les éléments qui ne sont pas points fixe par t sont dans l'orbite précédemment calculée.

```python
def estCycle(t):
    n=len(t); p=1
    # calcul d'un point non fixe
    while t[p]==p and p<n:
        p+=1
    # on elimine le cas particulier de l'identite
    if p==n:
        return False
    # calcul du cycle du point trouve
    s=p; u=[False]*n; u[s]=True
    while t[s]!=p:
        s=t[s]
        u[s]=True
    # verification de l'unicite de l'orbite
    for i in range(n):
        if t[i]!=i and not(u[i]):
            return False
    return True
```

Il a de nombreuses façons de répondre à cette question, mais l'avantage de la fonction ci-dessus est d'être de coût linéaire. Parmi les autres stratégies possibles (et coûteuses), citons par exemple :

- Déterminer un point non fixe s de la permutation (comme précédemment), puis utiliser la fonction `estDansOrbite` pour vérifier que tous les autres points non fixe sont dans l'orbite de s.
- Compter le nombre d'indices tels que $t[i] \neq i$ puis à vérifier que ce nombre est bien égal à la période de n'importe lequel de ces indices car les indices d'une même orbite ont la même période égale à la taille de l'orbite.

3 Opérations efficaces sur les permutations

10 Pour éviter d'appliquer la fonction `periode` à chaque indice de t, utilisons le fait que la période d'un indice est égale à la période de n'importe quel autre indice de son orbite. Parcourons les indices du tableau t à l'aide d'une boucle `for`. Pour chacun :

- si sa période est inconnue, on la calcule à l'aide de la fonction `periode` de la question 6 (coût linéaire en la période) puis l'orbite est parcourue pour fixer la période des éléments qui la compose (coût de nouveau linéaire en la période).
- Si la période de i est déjà connue, on passe à l'indice suivant.

Les périodes sont stockées dans un tableau `p` dont les éléments sont initialisés à 0. Ainsi si `p[i]` vaut zéro, c'est qu'on ne connaît pas la période de l'indice `i`.

On notera qu'une orbite est nécessairement parcourue deux fois. C'est malheureusement nécessaire car lorsqu'on la parcourt la première fois, on ne pas encore connaître sa taille avant de l'avoir parcourue en entier. Toutefois, de cette manière, le nombre d'opérations réalisées est linéaire en la somme des tailles des orbites, c'est-à-dire en la taille de t. En effet les orbites sont deux à deux disjointes et leur union est égale à E_n.

```python
def periodes(t):
    n=len(t); p=[0]*n
    for i in range(n):
        if p[i]==0:
            p[i]=periode(t,i)
            s = t[i]
            while s!=i:
                p[s]=p[i]
                s=t[s]
    return p
```

> La difficulté de cette question réside dans la contrainte de coût linéaire, ce qui nécessite d'avoir écrit la « bonne » fonction `periode` à la question 6, c'est-à-dire dont le coût n'excède pas la longueur de l'orbite.

11 La fonction reste est déjà définie en python : il suffit d'utiliser l'opérateur %. Pour le reste, c'est une application immédiate de l'énoncé. On commence par calculer le tableau des périodes. Une fois fait, il suffit de calculer pour chaque indice i son image par t^r où r est le reste de k modulo sa période.

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

12 Comme le signale l'énoncé, l'ordre d'une période est égale au plus petit commun diviseur des périodes de ces éléments. Pour trouver une permutation d'ordre strictement supérieur à sa taille, il suffit par exemple de prendre une permutation de taille 5, ayant deux orbites de 2 et 3 éléments. Considérons ainsi $t = [1, 0, 3, 4, 2]$. Alors,

$$t^2 = [0, 1, 4, 2, 3] \qquad t^3 = [1, 0, 2, 3, 4] \qquad t^4 = [0, 1, 3, 4, 2] \qquad t^5 = [1, 0, 4, 2, 3] \qquad \text{et} \qquad t^6 = [0, 1, 2, 3, 4]$$

Ainsi $\boxed{t = [1, 0, 4, 2, 3] \text{ est une permutation de taille 5 et d'ordre } 6 > 5.}$

13 L'énoncé demande d'écrire un algorithme de recherche du plus grand commun diviseur (pgcd) de deux entiers par la méthode d'Euclide que l'on peut résumer ainsi :

- on effectue la division euclidienne des deux nombres ;
- on recommence avec le diviseur et le reste ;
- et on s'arrête lorsque l'on obtient un reste nul.

Le pgcd des deux nombres est alors le dernier reste non nul dans la succession des divisions. Cette fonction peut s'écrire de manière immédiate sous forme récursive,

```
def pgcd(a,b):
    if a==0:
        return b
    else:
        return pgcd(b%a,a)
```

mais on peut aussi bien l'écrire en style impératif avec une boucle `while`.

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

14 Il suffit d'utiliser la fonction précédente et la formule donnée par l'énoncé.

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

15 Dans un premier temps, on calcule le tableau des périodes (coût linéaire). Ensuite, il n'y a plus qu'à effectuer le calcul du ppcm de toutes les périodes pour trouver l'ordre de la permutation. Remarquons toutefois que chaque élément d'une même orbite a la même période. Afin d'éviter de calculer plusieurs ppcm pour une même valeur de période, on utilise un tableau u pour recenser les périodes déjà rencontrées (une période est nécessairement un élément de $[\![1;n]\!]$ car la longueur d'un cycle ne peut excéder n). Avant de lancer un calcul de ppcm, on consulte le tableau u et on lance le calcul que si la période $p[i]$ n'a pas été rencontrée (auquel cas on met à jour le tableau u à la fin du calcul).

```
1 def ordreEfficace(t):
2     n=len(t); p=periodes(t)
3     u=[False]*n
4     s=1
5     for i in range(n):
6         if not(u[p[i]]):
7             s=ppcm(s,p[i])
8             u[p[i]]=True
9     return s
```