Sujet : X-ENS Informatique B 2017
Corrigé

Partie I. Une solution naïve en Python

Question 1.

```
def membre(p, Q):
    '''version puriste'''
    i, cont = 0, False
    n = len(Q)
    while i < n and not cont :
        if p[0] == Q[i][0]:
            if p[1] == Q[i][1]:
                cont = True
        i += 1
    return(cont)
```

La commande `element in sequence` n'était pas stipulée dans les rappels de PYTHON. Était-elle tolérée à la correction ? Je n'en sais rien. Même chose pour `Liste_1 == Liste_2` sans passer par le test élément par élément. La proposition de solution suivante ne rapportait pas forcément la totalité des points de la question.

```
def membre_bis(p, Q):
    '''version pythonesque'''
    for point in Q:
        if point == p:
            return(True)
        return(False)
```

Dans cette manière un peu puriste de voir les choses, je me suis forcé à proposer des algorithmes avec des boucles `while` sans utiliser la possibilité de quitter une fonction par un `return()` à l'intérieur d'une boucle `for`. Mais cette solution, peut être moins élégante, me semble tout à fait envisageable.

Question 2.

```
def intersection(P, Q):
    LI = []
    np, nq = len(P), len(Q)
    if np <= nq:
        for i in range(np):
            if membre(P[i], Q):
                LI.append(P[i])
    else :
        for j in range(nq):
            if membre(Q[j], P):
                LI.append(Q[j])
    return(LI)
```

Question 3.

Quelle que soit la différence de taille entre les ensembles P et Q de longueurs p et q, l'algorithme va parcourir chaque point de l'un puis chercher s'il est présent en le comparant au pire à chaque point de l'autre.

La complexité de la fonction précédente est donc au pire en $\mathcal{O}(p \times q)$.

Partie II. Une solution naïve en SQL

Question 4.

```
SELECT idensemble FROM Membre JOIN Points ON idpoint = id WHERE x = a AND y = b;
```

Question 5.

```
SELECT x, y FROM Points JOIN Membre AS Ei ON id = Ei.idpoint JOIN Membre AS Ej ON id = Ej.idpoint
WHERE Ei.idensemble = i AND Ej.idensemble = j;
```

Question 6.

```
SELECT DISTINCT id FROM Points JOIN Membre ON id = idpoint
WHERE idensemble IN (
SELECT idensemble FROM Membre JOIN Points ON id = idpoint WHERE x = a AND y = b)
AND id != (SELECT id FROM Points WHERE x = a and y = b);
```

Il était aussi possible d'utiliser une auto-jointure.

```
SELECT DISTINCT E2.idpoint FROM Membre AS E1 JOIN Membre AS E2
ON E1.idensemble = E2.idensemble
JOIN Points ON Points.id = E1.idpoint
WHERE x = 5 and y = 12
AND E2.idpoint != (SELECT id FROM Points WHERE x = 5 and y = 12);
```

Partie III. Codage de Lebesgue

Question 7.

Ici x vaut 1, donc $\overline{001}^2$ en binaire, et y vaut 6, soit $\overline{110}^2$ en binaire. Le codage binaire de Lebesgue est donc $\overline{010110}^2$, soit en base 4 avec la notation décimale : $\overline{112}^l$.

Le point (1, 6) sera donc représenté par son codage de Lebesgue en PYTHONpar la liste [1, 1, 2].

Question 8.

Voici une fonction bits(x, k) qui prend en argument deux entiers naturels x et k et qui renvoie la valeur du bit de coefficient 2^k dans la représentation binaire de x.

```
def bits(x, k):
    return ((x // pow(2, k)) % 2)
```

Cette fonction renvoie la valeur recherchée directement et non en faisant la conversion complète de x en binaire puis en prenant le chiffre de rang k, ce qui serait un algorithme naïf. Je ne vois pas pourquoi le sujet ne vous demandait pas cette petite fonction...

Voici alors la fonction demandée.

```
def code(n, p):
    res = []
    for i in range(n - 1, -1, -1):
        res.append(2 * bits(p[0], i) + bits(p[1], i))
    return(res)
```

Partie IV. Représentation d'un ensemble de points

Question 9.

$\overline{000}^l < \overline{012}^l < \overline{101}^l < \overline{233}^l < \overline{311}^l$.

Question 10.

```
def compare_pcodes(n, c1, c2):
    i = 0
    while i < n - 1 and c1[i] == c2[i]:
        i += 1
    if c1[i] < c2[i]:
        res = 1
    elif c1[i] > c2[i]:
        res = -1
    else:
        res = 0
    return(res)
```

Question 11.

La représentation binaire de l'ensemble de points S_1 vaut :
$\overline{S_1}^2 = \{(\overline{00}^2, \overline{00}^2), (\overline{11}^2, \overline{11}^2), (\overline{11}^2, \overline{10}^2), (\overline{01}^2, \overline{01}^2), (\overline{01}^2, \overline{10}^2), (\overline{10}^2, \overline{10}^2), (\overline{10}^2, \overline{11}^2)\}$.

On en déduit le codage de Lebesgue de chaque point.
$\overline{S_1}^l = \{\overline{00}^2\overline{00}^2, \overline{11}^2\overline{11}^2, \overline{11}^2\overline{10}^2, \overline{00}^2\overline{11}^2, \overline{01}^2\overline{10}^2, \overline{11}^2\overline{00}^2, \overline{11}^2\overline{01}^2\} = \{\overline{00}^l, \overline{33}^l, \overline{32}^l, \overline{03}^l, \overline{12}^l, \overline{30}^l, \overline{31}^l\}$.

Cet ensemble, une fois trié pour l'ordre lexicographique, s'écrit :
$\overline{S_1}^l = \{\overline{00}^l, \overline{03}^l, \overline{12}^l, \overline{30}^l, \overline{31}^l, \overline{32}^l, \overline{33}^l\}$.

Partie V. Calcul efficace de l'intersection d'ensembles de points

Question 12.

L'ensemble S_1, une fois compacté, s'écrit :
$\overline{S_1}^l = \{\overline{00}^l, \overline{03}^l, \overline{12}^l, \overline{34}^l\}$.

À la question précédente, le sujet demandait le code compacté. Or la notion de compactage n'était pas encore définie. Il faut donc ôter ce terme à la question 11.

Question 13.

```python
def ksuffixe(n, k, Q):
    test = True
    for i in range(n - 1, n - 1 - k, -1):
        if Q[i] != 4:
            test = False
    if test :
        return(Q[: n - 1 - k] + [4] * (k + 1))
    else:
        return(Q)
```

Question 14.

Voici une première fonction qui compare plusieurs éléments consécutifs d'une liste de codes de Lebesgue.

```python
def compare(n, i0, rep, L):
    '''Compare rep éléments consécutifs de L à partir du rang i0.
    L est une liste de codes de Lebesgue en base 4 de Dn x Dn.
    Renvoie True s'ils sont tous égaux, False sinon.'''
    indice, compt = i0, 2
    taille = len(L)
    test = compare_pcodes(n, L[indice], L[indice + 1]) == 0
    while compt < rep and test and indice < taille - 2:
        indice += 1
        compt += 1
        test = compare_pcodes(n, L[indice], L[indice + 1]) == 0
    return(test)
```

Voici une deuxième fonction qui compacte les blocs de 4 éléments consécutifs dans les codages de Lebesgue. J'utilise ici le fait que la fonction `ksuffixe(n, 0, S)` affecte le dernier élément de S à la valeur 4.

```python
def compacte_bloc(n, k, S):
    '''S est une liste de codages Lebesgue triée
    pour l'ordre lexicographique, et sans répétition.
    k est le rang à tester pour compacter les codages, donc de 1 à n.
    Renvoie la liste de codages compactés au rang k'''
    taille = len(S)
    res = []
    iS = 0    # iS parcourt S

    # On applique ksuffixe() à tous les éléments de S au rang précédent
    Smod = [ksuffixe(n, k - 1, si) for si in S]
    # Alors si un bloc de 4 éléments de rang k - 1 existe dans S
    # ils auront le même suffixe

    while iS < taille :   # Parcourt de tout S
        if iS < taille - 3 and compare(n, iS, 4, Smod):
            # si il reste plus de 4 éléments dans S
            # et si les 4 éléments suivants forme un même bloc
            res.append(Smod[iS])
            iS += 4
        else:
            res.append(S[iS])
            iS += 1
    return(res)
```

Ensuite, j'itère cette fonction de 1 à n compris avec la fonction suivante.

```python
def compacte(n, S):
    for k in range(n):
        S = compacte_bloc(n, k + 1, S)
        # S en argument ne sera pas modifié car le S de la boucle for
        # est une variable locale  e la fonction compacte()
    return(S)
```

Question 15.

```python
def compare_ccodes(n, P, Q):
    np, nq = len(P), len(Q)
    k, res= 0, 0
    while k < n and P[k] == Q[k] and P[k] != 4:
        # Parcourt des deux codes
        k += 1
    if P[k] < Q[k]:
        if Q[k] == 4:
            # P inclus dans Q
            res = 2
        else:
            # P inférieur à Q
            res = 1
    elif P[k] > Q[k]:
        if P[k] == 4:
            # Q inclus dans P
            res = -2
        else:
            # Q inférieur à P
            res = -1
    else:
        # dernier cas si P et Q ont même préfixe
        res = 0
    return(res)
```

C'est un procédé par disjonction de cas.

Question 16.

```python
def intersection2(n, P, Q):
    np, nq = len(P), len(Q)
    LI = [] # initialisation de la liste intersection
    kp, kq = 0, 0   # indices d'exploration des listes P et Q

    while kp < np and kq < nq:
        # variant : (np - kp) + (nq - kq)
        test = compare_ccodes(n, P[kp], Q[kq])
        if test == 0:         # P[kp] et Q[kq] égaux
            LI.append(P[kp])      # P[kp]=Q[kq] est dans l'intersection
            kp += 1               # Passage à P[kp + 1]
            kq += 1               # Passage à Q[kq + 1]
        elif test == 1:       # P[kp] < Q[kq]
            kp += 1               # Passage à P[kp + 1]
        elif test == -1:      # Q[kq] < P[kp]
            kq += 1               # Passage à Q[kq + 1]
        elif test == 2:       # P[kp] inclus dans Q[kq]
            LI.append(P[kp])      # P[kp] est dans l'intersection
            kp += 1               # Passage à P[kp + 1]
        elif test == -2:      # Q[kq] inclus dans P[kp]
            LI.append(Q[kq])      # Q[kq] est dans l'intersection
            kq += 1           # Passage à Q[kq + 1]
    return(LI)
```

À chaque itération de la boucle `while`, la fonction `compare_ccodes()` est appelée et `kp` ou `kq` est incrémenté, ou les deux.

Au pire des cas, la boucle `while` sera donc parcourue `len(P) + len(Q)` fois. Le nombre d'appels à la fonction `compare_ccodes()` effectués par `intersection2(n, P, Q)` sera donc en $\mathcal{O}($`len(P) + len(Q)`$)$.