Corrigé : intersection de deux ensembles de points (X PC 2017)

Partie I. Une solution naïve en Python

Question 1.

```
def membre(p, q):
    for x in q:
        if x == p:
            return True
    return False
```

Question 2.

```
def intersection(p, q):
    inter = []
    for x in p:
        if membre(x, q):
            inter.append(x)
    return inter
```

Question 3. Notons $|p|$ et $|q|$ les longueurs respectives des listes p et q. La fonction `membre` réalise un nombre de comparaisons au plus égal à $|q|$. La fonction `intersection` utilise cette fonction un nombre de fois égal à $|p|$, donc la complexité totale de cette fonction est en $O(|p| \cdot |q|)$.

Partie II. Une solution naïve en SQL

Question 4.

```
SELECT idensemble FROM points JOIN membres ON id = idpoint WHERE x = a AND y = b
```

Question 5.

```
SELECT id FROM points JOIN membres as m1 ON id = m1.idpoint
                     JOIN membres as m2 ON id = m2.idpoint
WHERE m1.idensemble = i AND m2.idensemble = j
```

Question 6. Une solution consiste à utiliser le résultat de la question 4 comme sous-requête (mais d'autres rédactions sont possibles).

```
SELECT idpoint FROM membres
WHERE idensemble IN (SELECT idensemble FROM points JOIN membres ON id = idpoint
                     WHERE x = a AND y = b)
```

Partie III. Codage de Lebesgue

Question 7. Pour $n = 3$ le codage de Lebesgue du point $(1,6) = (\overline{001}^2, \overline{110}^2)$ est $\overline{01}^2\overline{01}^2\overline{10}^2 = \overline{112}^\ell$. Ce point est donc représenté en Python par la liste `[1, 1, 2]`.

Question 8.

```
def code(n, p):
    [x, y] = p
    lebesgue = []
    for k in range(n - 1, -1, -1):
        lebesgue.append(2 * bits(x, k) + bits(y, k))
    return lebesgue
```

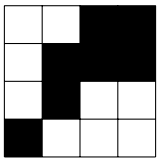
Partie IV. Représentation d'un ensemble de points

Question 9. On a $\overline{000}^{\ell} < \overline{012}^{\ell} < \overline{101}^{\ell} < \overline{233}^{\ell} < \overline{311}^{\ell}$.

Question 10.

```
def compare_pcodes(n, c1, c2):
    for i in range(n):
        if c1[i] < c2[i]:
            return 1
        elif c1[i] > c2[i]:
            return -1
    return 0
```

Question 11. On peut bien sûr procéder en calculant le codage de Lebesgue de chacun des points puis en les triant par ordre lexicographique, mais il est préférable de se familiariser avec la représentation par cadran qui sera utilisée plus tard. Représentons-donc les points sur un dessin :



Il y a deux points dans le cadran 0, et au sein de ce cadran ces deux points se trouvent dans les cadrans 0 et 3. les deux premiers points de la liste sont donc `[0, 0]` et `[0, 3]`.
Il y a un point dans le cadran 1 et au sein de ce cadran ce point se trouve dans le cadran 2. Le point suivant est donc `[1, 2]`.
Il n'y a pas de point dans le cadran 2, et quatre dans le cadran 3. Au sein de ce cadran les points se trouvent dans chacun des quatre cadrans donc ils sont représentés par `[3, 0]`, `[3, 1]`, `[3, 2]` et `[3, 3]`.
L'ensemble S_1 est donc représenté sous forme de codage de Lebesgue par la liste :

`[[0, 0], [0, 3], [1, 2], [3, 0], [3, 1], [3, 2], [3, 3]]`

Partie V. Calcul efficace de l'intersection d'un ensemble de points

Question 12. Dans l'exemple précédent, les quatre derniers points remplissent tout le cadran 3 donc peuvent être représentés par le couple `[3, 4]`. L'AQL de l'ensemble S_1 est donc :

`[[0, 0], [0, 3], [1, 2], [3, 4]]`

Question 13.

```
def ksuffixe(n, k, q):
    qq = list(q)
    for i in range(k):
        if qq[n - 1 - i] != 4:
            return qq
    qq[n - 1 - k] = 4
    return qq
```

Question 14. Voici une question bien difficile ! La fonction ci-dessous repose sur l'observation suivante : à l'étape k, un quadrant complet de côté 2^{k+1} est présent dans P dès lors que les quatre codages compactés `ksuffixe(n, k, aql[i])`, `ksuffixe(n, k, aql[i + 1])`, `ksuffixe(n, k, aql[i + 2])` et `ksuffixe(n, k, aql[i + 3])` sont égaux. Néanmoins, puisqu'ils sont classés par ordre lexicographique, il suffit de comparer le premier et le quatrième d'entre eux pour que cette condition soit réalisée. Dans ce cas de figure, on réalise le compactage (lignes 12-14) ; dans le cas contraire (pas de compactage possible), on se contente de recopier tel quel le code (lignes 16-17).

```
1  def compacte(n, s):
2      aql = list(s)
3      for k in range(n):
4          temp = []
5          i = 0
6          while i < len(aql):
7              b = True
8              if i + 3 < len(aql):
9                  c1 = ksuffixe(n, k, aql[i])
10                 c2 = ksuffixe(n, k, aql[i + 3])
11                 if compare_pcodes(n, c1, c2) == 0:
12                     temp.append(c1)
13                     i += 4
14                     b = False
15             if b:
16                 temp.append(aql[i])
17                 i += 1
18         aql = temp
19     return aql
```

Question 15.

```
def compare_ccodes(n, c1, c2):
    for i in range(n):
        if c1[i] < c2[i]:
            if c2[i] < 4:
                return 1
            else:
                return 2
        elif c1[i] > c2[i]:
            if c1[i] < 4:
                return -1
            else:
                return -2
    return 0
```

Question 16. On parcourt conjointement les deux AQL p et q à l'aide des indices i et j.

- Lorsque $p[i] = q[j]$ (lignes 6-9) cet élément commun est ajouté à l'intersection et les indices i et j augmentés ;
- lorsque $p[i] < q[j]$ (lignes 10-11) il est certain (puisque les listes sont classées par ordre croissant) que $p[i]$ ne peut appartenir à l'intersection ;
- lorsque $q[j] < p[i]$ (lignes 12-13) il est de même certain que $q[j]$ ne peut appartenir à l'intersection ;
- lorsque $p[i] \subsetneq q[j]$ (lignes 14-16), $p[i]$ est ajouté dans l'intersection ;
- enfin, lorsque $q[j] \subsetneq p[i]$ (lignes 17-19), $q[j]$ est ajouté dans l'intersection.

```
1  def intersection2(n, p, q):
2      inter = []
3      i, j = 0, 0
4      while i < len(p) and j < len(q):
5          c = compare_ccodes(n, p[i], q[j])
6          if c == 0:
7              inter.append(p[i])
8              i += 1
9              j += 1
10         elif c == 1:
11             i += 1
```

```
12          elif c == -1:
13              j += 1
14          elif c == 2:
15              inter.append(p[i])
16              i += 1
17          elif c == -2:
18              inter.append(q[j])
19              j += 1
20      return inter
```