Épreuve d'informatique X/ENS - 2017 - MP/PC/PSI

Intersection de deux ensembles de points

Partie I. Une solution naïve en PYTHON

Question 1 : On commence par écrire une fonction `egal p1 p2` qui retourné `True` si les points `p1` et `p2` sont égaux et `False` sinon.
Dans la fonction `membre p q`, on teste l'égalité entre le point `p` et les éléments de la liste de points `q` dans la boucle des lignes 5 à 7. S'il y a égalité, on quitte le programme et on retourne `True` (ligne 7). Sinon, on retourne `False` à la ligne 8.

```
1 def egal(p1,p2):
2     return(p1[0]==p2[0] and p1[1]==p2[1])
3
4 def membre(p,q):
5     for i in range(len(q)):
6         if egal(p,q[i]):
7             return(True)
8     return(False)
```

Question 2 : Dans la boucle des lignes 3 à 5, on teste l'appartenance de chaque élément de la liste de points `p` à la liste de points `q` par un appel à la fonction `membre`. Si le point `p[i]` appartient à la liste de points `q`, on l'ajoute à la liste résultat `inter`. On retourne le résultat à la ligne 6.

```
1 def intersection(p,q):
2     inter = []
3     for i in range(len(p)):
4         if membre(p[i],q):
5             inter.append(p[i])
6     return(inter)
```

Question 3 : Dans le pire des cas, c'est-à-dire, quand les ensembles `p` et `q` sont d'intersection vide, le nombre de comparaisons pour chaque appel à la fonction `membre` est égal à 2 fois la longueur de `q`. Le nombre d'appels à la fonction `membre` étant égal à la longueur de `p`, la complexité de l'algorithme est en $O(|p|\ |q|)$ où $|p|$ et $|q|$ sont les longueurs respectives des listes `p` et `q`.

conclusion : On a une complexité en $O(|p|\ |q|)$

Partie II. Une solution naïve en SQL (PSI*)

Question 4 : On effectue une jointure sur les tables `points` et `membre` avec la condition d'égalité sur l'identifiant du point. Dans la table obtenue, on sélectionne les ensembles qui contiennent le point de coordonnées (a, b).

```
1 SELECT idensemble
2     FROM points JOIN membre ON id = idpoint
3     WHERE x = a AND y = b;
```

Question 5 : On effectue une jointure sur les tables `points`, `membre` et `membre` avec la condition d'égalité sur l'identifiant des points entre les tables `points` et `membre`. On sélectionne les coordonnées (x, y) des points qui appartiennent à l'intersection des ensembles i et j.

```
1 SELECT x,y FROM points JOIN membre ens1 JOIN membre ens2
2             ON id = ens1.idpoint AND id = ens2.idpoint
3                     WHERE ens1.idensemble = i AND ens2.idensemble = j;
```

Question 6 : La formule entre parenthèses permet de créer la table des identifiants des ensembles qui contiennent le point de coordonnées (a, b). Elle permet de générer la table des identifiants des points appartenant à au moins l'un de ses ensembles. La partie « `GROUP BY idpoint` » permet de n'afficher qu'une seule fois les identifiants.

```
1 SELECT idpoint FROM membre
2     WHERE idensemble IN
3         (SELECT idensemble FROM points JOIN membre ON id = idpoint
4                 WHERE x=a and y=b)
5     GROUP BY idpoint;
```

Partie III. Codage de Lebesgue

Question 7 : le couple $(1, 6)$ s'écrit en binaire $(\overline{001}^2, \overline{110}^2)$ pour $n = 3$. Le codage entremêlé nous donne $\overline{010110}^2$ que l'on découpe en $\overline{01}^2\overline{01}^2\overline{10}^2$ ce qui nous donne en base 4 : 112.

conclusion : Le codage de Lebesgue du point $(1, 6)$ est la liste $[1, 1, 2]$ pour $n = 3$

Question 8 : On commence par récupérer les coordonnées x et y du point p (ligne 2 et 3). On crée la liste représentant le codage de Lebesgue du point p en partant de la liste vide (ligne 4) et en ajoutant successivement les éléments de la liste dans la boucle des lignes 5 et 6. On effectue une boucle descendante pour aller des bits de poids forts aux bits de poids faibles.

```
1 def code(n,p):
2     x = p[0]
3     y = p[1]
4     codage = []
5     for k in range(n-1,-1,-1):
6         codage.append(2*bits(x,k)+bits(y,k))
7     return(codage)
```

Partie IV. Représentation d'un ensemble de points

Question 9 : $\boxed{\overline{000}^{\ell} < \overline{012}^{\ell} < \overline{101}^{\ell} < \overline{233}^{\ell} < \overline{311}^{\ell}}$

Question 10 : On parcourt simultanément les codages de Lebesgue `c1` et `c2`. Pour la première valeur de k telle que `c1[k]`$\neq$`c2[k]`, on retourne la valeur 1 si `c1[k]`$<$`c2[k]`, c'est-à-dire si `c1` $<$ `c2` ou -1 si `c1` $>$ `c2`.
Si on parcourt la boucle sans quitter le programme, c'est que les deux codages sont identiques, on retourne la valeur 0 (ligne 7).

```
1 def compare_pcodes(n,c1,c2):
2     for k in range(n):
3         if c1[k] < c2[k]:
4             return(1)          # cas c1 < c2
5         elif c1[k] > c2[k]:
6             return(-1)         # cas c2 < c1
7     return(0)                  # cas c1 = c2
```

Question 11 : L'ensemble des points $\{(0,0),(3,3),(3,2),(1,1),(1,2),(2,2),(2,3)\}$, se code en binaire pour $n=2$, en

$$\{(\overline{00}^2,\overline{00}^2),\ (\overline{11}^2,\overline{11}^2),\ (\overline{11}^2,\overline{10}^2),\ (\overline{01}^2,\overline{01}^2),\ (\overline{01}^2,\overline{10}^2),\ (\overline{10}^2,\overline{10}^2),\ (\overline{10}^2,\overline{11}^2)\}$$

Ce qui nous donne le codage de Lebesgue de chaque point :
$\{(\overline{00}^2,\overline{00}^2),\ (\overline{11}^2,\overline{11}^2),\ (\overline{11}^2,\overline{10}^2),\ (\overline{00}^2,\overline{11}^2),\ (\overline{01}^2,\overline{10}^2),\ (\overline{11}^2,\overline{00}^2),\ (\overline{11}^2,\overline{01}^2)\}$
$= \{\overline{00}^{\ell},\ \overline{33}^{\ell},\ \overline{32}^{\ell},\ \overline{03}^{\ell},\ \overline{12}^{\ell},\overline{30}^{\ell},\ \overline{31}^{\ell}\}$
Cet ensemble, une fois trié pour l'ordre lexicographique, s'écrit :

$$\boxed{\{\overline{00}^{\ell},\overline{03}^{\ell},\overline{12}^{\ell},\overline{30}^{\ell},\overline{31}^{\ell},\ \overline{32}^{\ell},\ \overline{33}^{\ell}\}}$$

représenté par la liste de listes : $[[0,0],[0,3],[1,2],[3,0],[3,1],[3,2],[3,3]]$

Partie IV. Trier des listes partiellement triées

Question 12 Les points $\overline{30}^{\ell},\overline{31}^{\ell},\ \overline{32}^{\ell},\ \overline{33}^{\ell}$ appartiennent à un même quadrant et on ne peut pas effectuer d'autres regroupements. L'ensemble S_1 est compacté sous la forme $\{\overline{00}^{\ell},\overline{03}^{\ell},\overline{12}^{\ell},\overline{34}^{\ell}\}$.

conclusion : $\boxed{\text{L'AQL de l'ensemble } S_1 \text{ est la liste } [[0,\ 0],\ [0,\ 3],\ [1,\ 2],\ [3,\ 4]]}$

Question 13 : On suppose que q est une liste recevable, c'est-à-dire que si $q[n-k-1]=4$ alors $q[i]=4$ pour $i \in [\![n-k-1, n-1]\!]$.
Si $k=0$, il suffit de retourner la liste de départ en remplaçant le dernier élément par 4.
Si $k>0$, on vérifie que l'élément d'indice $n-k$ est un 4. Si c'est le cas, on retourne la copie de la liste de départ en ayant remplacé l'élément d'indice $n-k-1$ par 4, sinon on retourne la copie de la liste de départ.

```
1 def ksuffixe(n,k,q):
2     nouv_q = list(q)
3     if k == 0:
4         nouv_q[n-1] = 4
5     elif q[n-k] == 4 :
6         nouv_q[n-k-1] = 4
7     return(nouv_q)
```

Question 14 : On effectue des opérations de compactage successives à l'aide de la boucle conditionnelle des lignes 5 à 21. On s'arrête dès qu'une opération de compactage n'a pas permis de réduire la taille de la liste. La k-ième opération de compactage s'effectue dans la boucle conditionnelle des lignes 9 à 16.
On détecte un compactage possible si les éléments d'indice i et $i+3$ appartiennent au même quadrant (lignes 10-11). Si c'est le cas, on ajoute la représentation du quadrant à la liste résultat et on passe à l'élément d'indice $i+4$ (lignes 12 et 13). Dans le cas contraire, on ajoute l'élément d'indice i à la liste résultat et on passe à l'élément d'indice $i+1$ (ligne 14 à 16).

```
1  def compacte(n,s):
2      k=0
3      bool = True
4      res = list(s)
5      while k<n and bool:
6          i = 0
7          tmp = []
8          m = len(res)
9          while i < m:
10             if i+3 < m and compare_pcodes(n,ksuffixe(n,k,res[i]),
11                                           ksuffixe(n,k,res[i+3])) == 0:
12                 tmp.append(ksuffixe(n,k,res[i]))
13                 i += 4
14             else :
15                 tmp.append(res[i])
16                 i += 1
17         if len(res) == len(tmp):
18             bool = False      # il n'y a pas eu de compactage
19         else :                # il y a eu compactage
20             res = list(tmp) # on travaille sur la nouvelle liste
21             k += 1          # on passe à l'étape suivante
22     return(res)
```

Question 15 : On adapte la fonction de comparaison de la question 10. On parcourt simultanément les codages de Lebesgue des parties `p` et `q`. Pour la première valeur de k telle que `p[k]`$\neq$`q[k]`, on a une inclusion si l'une des deux valeurs vaut 4 (lignes 6 à 9) sinon, soit les deux parties sont égales si `p[k]` = `q[k]` = 4 (ligne 5), soit les ensembles sont disjoints dans le cas contraire (lignes 12 à 15).

```
1  def compare_ccodes(n,p,q):
2      k = 0
3      while k < n:
4          if p[k] == 4 and q[k] == 4:    # cas p = q
5              return(0)
6          elif p[k] == 4 and q[k] != 4: # cas q inclus dans p
7              return(-2)
8          elif p[k] != 4 and q[k] == 4: # cas p inclus dans q
9              return(2)
10         elif p[k] == q[k] :
11             k += 1
12         elif  p[k] < q[k] :            # p et q disjoints
13             return(1)                  # p < q
14         else :
15             return(-1)                 # q < p
16     return(0)    # p = q
```

Question 16 : Pour calculer l'intersection des ensembles représentés par les listes `p` et `q`, on parcourt simultanément les listes p et q. La boucle conditionnelle des lignes 4 à 18 s'arrête dès que l'on a parcouru entièrement l'une des deux listes.

```
1  def intersection(n,p,q):
2      lenp,lenq = len(p),len(q)
3      i,j = 0,0; inter = []
4      while i < lenp and j < lenq:
5          c = compare_ccodes(n,p[i],q[j])
6          if  c == 1:
7              i += 1      # p[i] < q[j], on incrémente i
8          elif c == -1:
9              j += 1      # p[i] > q[j], on incrémente j
10         elif c == 2:   # cas p[i] inclus dans q[j],
11             inter.append(p[i]) # on ajoute p[i] à la liste résultat
12             i += 1             # et on incrémente i
13         elif c == -2:  # cas p[j] inclus dans p[i],
14             inter.append(q[j]) # on ajoute q[j] à la liste résultat
15             j += 1             # on incrémente j
16         else :         # c == 0, p[i] = q[j],
17             inter.append(p[i]) # on ajoute p[i] à la liste résultat
18             i,j = (i+1,j+1)    # on incrémente i et j
19     return(inter)
```