LYCEE DAUDET – LYCEE JOFFRE

CONCOURS BLANC 2021 **Filière MPSI**

SOLUTION DE LA COMPOSITION D'INFORMATIQUE DE TRONC COMMUN

1. Suivant la recommandation de l'énoncé, on peut (mais cela n'est nullement nécessaire sinon) créer une fonction auxiliaire testant l'égalité de deux points :

```
def egalP(p1,p2):
    return (p1[0]==p2[0] and p1[1]==p2[1])
```

Pourquoi cela n'est-il pas nécessaire ? Car python gère bien sûr directement l'égalité `p1==p2`, mais le paragraphe **Rappel de PYTHON** de l'énoncé incite à être prudent et à n'utiliser que des opérations élémentaires, même s'il ne parle pas du test d'égalité.
Pour la fonction `membre` on parcourt la liste `Q` dont chaque entrée est un point et on teste l'égalité de chacune de ses entrées avec le point `p`, le `return True` force la sortie de la fonction donc interrompt la boucle `for` si on trouve une entrée égale à `p`. Si la boucle `for` arrive à son terme c'est que `p` n'est pas dans `Q`.

```
def membre(p,Q):
    for i in range(len(Q)):
        if egalP(Q[i],p):
            return True
    return False
```

Variante possible : on peut parcourir les *valeurs* de la liste `Q` plutôt que ses indices, ce qui donne :

```
def membrebis(p,Q):
  for q in Q:
      if egalP(p,q):
          return True
  return False
```

2. L'algorithme est donné par la question : on itère sur tous les points `p` de `P` les appels à la fonction `membre(p,Q)`.

```
def intersection(P,Q):
  L=[]
  for p in P:
      if membre(p,Q):
          L.append(p)
  return L
```

3. La fonction `membre` a une complexité en $O(len(Q))$ puisqu'elle parcourt au plus toute la longueur de `Q` et fait deux tests d'égalité à chaque tour. La fonction `intersection` fait exactement $len(P)$ tours de boucle et à chaque tour un appel de la fonction `membre(p,Q)` et au plus un `append` (considéré en $O(1)$), donc la complexité à chaque tour est $O(len(Q))$ et la complexité totale en $O(len(P).len(Q))$.

4. Il suffit de faire une jointure entre la table `POINTS` qui contient les coordonnées des points et la table `MEMBRE`.

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

ou bien (ancienne syntaxe pour la jointure, correcte mais moins recommandée, suivant l'idée de la sélection dans le produit cartésien mais en fait optimisée comme le `JOIN`) :

```
SELECT idensemble FROM membre,  points
WHERE  id = idpoint AND x = a AND y = b;
```

5. Plusieurs méthodes possibles.
Méthode 1 : avec la commande `INTERSECT` de `SQL` qui paraît naturelle ici ! On constitue la table des points de l'ensemble i et celle de l'ensemble j et INTERSECT :

```
(SELECT x,y FROM POINTS JOIN MEMBRE ON idpoint=id WHERE idensemble=i ) INTERSECT
(SELECT x,y FROM POINTS JOIN MEMBRE ON idpoint=id WHERE idensemble=j)
```

N.B. Les parenthèses sont mises ici pour la clarté, mais en SQLite en tout cas, il vaut mieux ne pas les mettre, donc ne seront pas comptées.
Méthode 2 : pour les fans de jointure ! La première jointure donne un tableau avec sur chaque ligne un point et un ensemble qui le contient, de même la seconde jointure rajoute encore un ensemble qui contient ce même point, et enfin les conditions de sélection ne gardent que les lignes où le premier ensemble a pour identifiant i et le second j.

```
SELECT x, y FROM points
JOIN membre M1 ON id = M1.idpoint              -- Le point est dans le premier ensemble.
JOIN membre M2 ON id = M2.idpoint              -- Le point est dans le deuxième ensemble.
WHERE M1.idensemble = i AND M2.idensemble = j; -- Ce sont les bons.
```

6. On utilise une sous-requête donnant les `idensemble` des ensembles contenant le point (a, b).

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

7. On sait que l'écriture en binaire sur 3 bits de 1 est $\overline{\mathbf{001}}^2$ et celle de 6 est $\overline{110}^2$. Donc le codage de Lebesgue de $(1, 6)$, obtenu en entrelaçant ces deux écriture est $(\overline{010110}^2)$.
En regroupant les bits deux par deux $\overline{01}^2\overline{01}^2\overline{10}^2$ ce qu'on écrit en base 4 pour obtenir le codage demandé : $\boxed{\overline{112}^\ell}$.

8. On note `x,y` les deux entrées de `p` On fait des appels successifs à la fonction `bits` dans l'ordre des k décroissants en partant du bit de poids fort i.e. représentant le coefficient de 2^{n-1} dans l'écriture binaire. On utilise la syntaxe du `range` décroissant en `python`.

```
def code(n,p):
    x=p[0]
    y=p[1]
    L=[]
    for k in range(n-1,-1,-1):
        xk=bits(x,k)
        yk=bits(y,k)
        L.append(2*xk+yk)
    return L
```

Bonus si on veut programmer la fonctions `bits` à la maison :
N.B. Cette déclaration de fonction `bits(x,k)` a un défaut : elle ne prend pas la variable `n`, qui doit donc être déclarée en global.

```
def base2(x,n):
    "renvoie une liste donnant l'écriture en base 2 de x sur une liste de longueur n"
    liste=[]
    if x==0:
        liste=[0] # Cas à mettre à part car n'entre pas dans la boucle qui suit
    else :
        while (x>0):
            a=x%2
            liste=[a]+liste # on rajoute a du bon côté...
            x=x//2 # quotient de la div. eucl.
    l=len(liste)
    return [0]*(n-l)+liste

def bits(x,k):
    """ Pour  cette fonction le n doit être une variable globale, c'est pas terrible"""
    return base2(x,n)[-k-1]
```

9. On commence donc par comparer les bits de poids forts (le premier bit) et s'ils sont égaux on regarde le deuxième etc. On obtient donc :

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

On notera que cet ordre n'est pas surprenant !
Barème ? 1 pt.

10. Pas de difficulté particulière :

```
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
```

Remarque : l'opérateur de comparaison < sur les listes de nombres en Python correspond exactement à cet ordre, donc en fait on aurait pu mettre simplement :

```
def compare_pcodes_bis(n,c1,c2):
  if c1<c2:
      return 1
  elif c1==c2:
      return 0
  else :
      return -1
```

mais cette solution « paresseuse » n'est pas très efficace puisqu'elle occasionne deux parcours de chaque liste si $c1 \geq c2$.

11. **Méthode 1** On suit les mêmes étapes que pour S_0 dans le sujet :
On commence par écrire ces points en binaire :

$$\overline{S_1}^2 = \{(\overline{00}^2, \overline{00}^2), (\overline{11}^2, \overline{11}^2), (\overline{11}^2, \overline{10}^2), (\overline{01}^2, \overline{01}^2), (\overline{01}^2, \overline{10}^2), (\overline{10}^2, \overline{10}^2), (\overline{10}^2, \overline{11}^2)\}$$

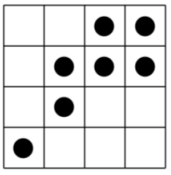
puis on entrelace pour avoir le codage de Lebesgue d'abord en base 2 puis en base 4 :

$$\overline{S_1}^\ell = \{\overline{00}^2\overline{00}^2, \overline{11}^2\overline{11}^2, \overline{11}^2\overline{10}^2, \overline{00}^2\overline{11}^2, \overline{01}^2\overline{10}^2, \overline{11}^2\overline{00}^2, \overline{11}^2\overline{01}^2\} = \{\overline{00}^\ell, \overline{33}^\ell, \overline{32}^\ell, \overline{03}^\ell, \overline{12}^\ell, \overline{30}^\ell, \overline{31}^\ell\}$$

La représentation triée par ordre lexicographique sera donc (les accolades données dans le sujet sont maladroite, car un ensemble n'est pas ordonné) :

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

Méthode 2 : on fait la représentation graphique :



et on identifie les points en fonctions des numéros des carrés imbriqués auxquels ils appartiennent.

12. Sur le dessin fait dans la réponse à la question 11 On peut donc compacter le carré en haut à droite i.e. le carré numéro 3, ce qui donne 34 et le codage compacté :

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

13. On commence par parcourir `q` suivant un range décroissant pour les k dernières entrées à la recherche d'une entrée qui n'est pas un 4 : si on en trouve une, on renvoie `q`.
Sinon, on crée une copie `L` de `q` via `L=list(q)`, et on modifie les $k+1$ dernières entrées de `L` pour y mettre un 4 et on renvoie `L`. Ceci car on ne doit pas modifier la liste `q`.

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

14. *Voyons comment l'algorithme doit tourner sur l'exemple de la figure 3 :*
L'énoncé dit de faire une boucle sur k : • pour $k = 0$ on doit remplacer les quadrants complets de côté $2^{0+1} = 2$ par leur représentation compactée avec un 4 à la fin. On remarque pour cela que la fonction `ksuffixe(n,k,q)` définie à la question précédente, si on lui passe l'argument $k = 0$ remplacera toujours la dernière entrée de `q` par un 4.
Donc au premier tour de boucle on doit tester s'il y a 4 entrées q_i de s qui ont le même $\texttt{ksuffixe}(\texttt{n}, 0, \texttt{q}_\texttt{i})$ et si oui, on les remplace pour la valeur de ce `ksuffixe`. Dans l'exemple de la figure 3, c'est ce qui se passe pour les trois carrés pleins de taille 2×2 que l'on voit bien, ainsi que les quatre carrés pleins de taille 2×2 qui sont dans le carré plein de taille 4×4.
• pour $k = 1$ on recommence pour les quadrant complets de côté 4 leurs éléments sont déjà des compactés de carrés de côté 2 et voilà.

D'une manière générale : la fonction `ksuffixe` permet d'identifier à quel bloc appartient `q` : au tour $k = 0$, elle donnera le même résultat pour chaque élément d'un des quatre gros blocs. Comme la liste `s` est ordonnée pour l'ordre lexicographique, si on regarde le premier élément `s[0]` de la liste `s`, pour savoir s'il est compactable, il suffit de savoir si le 4ème élément `s[3]` a le même suffixe `ksuffixe(n,0,s[3])` et si oui, on compacte cette élément dans une liste `temp` et on examine ensuite directement `s[4]`, autrement dit on fait des pas de 4. Sinon, on recopie tel quel l'élément s[0] dans la liste définitive et on fait un pas de 1.
En pratique, on va gérer deux listes en plus de `s`. L'avantage de `ksuffixe` est que les données qui n'ont pas été compactées ne seront plus modifées.

```
def compacte(n,s):
    L=list(s) # la liste qu'on renverra à la fin
    for k in range(n):
        M=[] # une liste qui va contenir
        # le résultat de la transformation à l'étape k
        # qu'on reversera dans L
        i=0
        while i <len(L):
            # on parcourt les cases de L
            carre_courant=ksuffixe(n,k,L[i])
            if i+3<len(L) and ksuffixe(n,k,L[i+3])==carre_courant:
                M.append(carre_courant)
                i=i+4
            else :
                M.append(L[i]) # cas non compacté,
                # ces gens ne seront plus modifiés par ksuffixe au tour suivant
                i=i+1
        L=list(M)
    return L
```

15.
```
def compare_ccodes(n,p,q):
    i=0
    while i<n and p[i]==q[i]:
        i=i+1
    if i==n:
        return 0
    if p[i]<q[i]:
        if q[i]<4:
            return 1
        else :
            return 2
    if p[i]>q[i]:
        if p[i]<4:
            return -1
        else:
            return -2
```

On parcourt les listes `p` et `q` de gauche à droite.
• si `p[0]!= q[0]` on sait que les listes `p,q` représentent deux quadrants qui sont dans un grand carré différent donc sont disjointes. et donc on peut les comparer pour l'ordre lexicographique en comparant $p[0]$ et $q[0]$ premier élément.
• sinon on continue tant que `p[i]==q[i]` et au premier moment où `p[i]!=q[i]` s'il existe, on regarde non seulement si `p[i]<q[i]` ou l'inverse, mais aussi si le plus grand des deux fait 4 ce qui donnerait une inclusion.

16. La base est un algorithme plus classique de fusion de listes triées, on compare à chaque fois les éléments en tête des deux listes, on garde les éléments communs pour avoir l'intersec-

tion. Ce qui est intéressant ici en plus est qu'avec la fonction de la question précédente, si `compare_ccodes(n,p[i],q[j])` renvoie 1 ou -1, on est sûr que `p[i]` et `q[j]` sont dans des quadrants disjoints, celui de `q[j]` étant strictement plus grand que celui de `p[i]` et comme les éléments de `q` sont triés, on est sûr qu'il n'y aura aucun élément `q[k]` de `q` avec $k \geq j$ qui sera égal à `p[i]` . On est donc sûr que `p[i]` n'est pas dans l'intersection.

```
def intersectionRusee(n,p,q):
    i=0 # indice de l'entrée courante de p
    j=0 # indice de l'entrée courante de q
    F=[]# la liste intersection/fusion
    while i <len(p) and j<len(q):
        C=compare_ccode(n,p[i],q[i])
        if C==0:
            F.append(p[i])
            i=i+1
            j=j+1
        elif C==1:
            i=i+1 # p[i] n'est pas dans l'intersection
        elif C==-1:
            j=j+1
        elif C==2:
            F.append(p[i])
            i=i+1
        elif C==-2:
            F.append(q[j])
            j=j+1
```