

1 Une solution naïve en PYTHON

Je définis une fonction qui teste l'égalité entre deux listes :

```
def EGAL(a,b):
    n=len(a)
    if n!=len(b):
        return False
    i=0
    while i<n and a[i]==b[i]:
        i+=1
    return i==n

1. def membre(p,q):
    for i in range(len(q)):
        if EGAL(q[i],p):
            return True
    return False

2. def intersection (p,q):
    res=[]
    for i in range(len(p)):
        if membre(p[i],q):
            res.append(p[i])
    return res
```

3. Lors de l'exécution de `membre(p,q)` il y a au plus $2 \cdot \text{len}(q)$ tests (2 par tours de boucle).
Lors de l'exécution de `intersection (p,q)` il y a $\text{len}(p)$ tours de boucles.
La complexité de l'exécution de `intersection (p,q)` est $\mathcal{O}(\text{len}(p)\text{len}(q))$.

2 Une solution naïve en SQL

4. `SELECT idensemble FROM MEMBRE JOIN POINTS ON idpoint=id WHERE x=a AND y=b`

5. `(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)`

On peut également stocker le résultat de la requête suivante dans une sous-table :

`ENSEMB1=(SELECT id,x,y FROM POINTS JOIN MEMBRE ON id=idpoint WHERE idensemble=i)`

puis :

`SELECT x,y FROM ENSEMB1 JOIN MEMBRE ON id=idpoint WHERE idensemble=j`

6. `SELECT idpoint FROM MEMBRE WHERE idensemble IN`

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

ou

`SELECT M1.idpoint FROM (MEMBRE AS M1) WHERE EXISTS`

`(SELECT * FROM (MEMBRE AS M2) JOIN POINTS ON M2.idpoint=id`

`WHERE x=a AND y=b AND M1.idensemble=M2.idensemble )`

ou pour les orthodoxes

`SELECT M1.idpoint FROM (MEMBRE AS M1) JOIN`

`(SELECT M2.idensemble AS id2 FROM (MEMBRE AS M2) JOIN POINTS ON M2.idpoint=id WHERE x=a AND y=b)`

`ON M1.idensemble=id2`

3 Codage de Lebesgue

7. Pour $n = 3$, on a $1 = \overline{001}^2$ et $6 = \overline{110}^2$.

Le point $(1, 6)$ admet alors comme codage de Lebesgue $\overline{010110}^2$ représenté par le nombre : $\overline{01}^2\overline{01}^2\overline{10}^2$ ce qui s'écrit $\overline{112}^\ell$.

Pour $n = 3$, la liste PYTHON `[1,1,2]` représente le codage de Lebesgue de $(1, 6)$.

8. On lit successivement les bits de x et y en commençant par les bits de poids forts.

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

4 Représentation d'un ensemble de points

9. Le tri des codages suivants par ordre croissant pour l'ordre lexicographique :

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

10. On compare le bit de poids fort.

S'il y a inégalité stricte, on peut conclure sinon on passe au bit suivant. Si on arrive à la fin de la liste, en cas d'égalité.

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

11. On obtient la figure suivante (je traite la question en enlevant le mot « compacté ») :

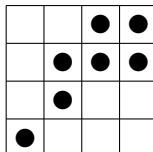


Figure c1 - Ensemble des points S_1 à compacter.

Par lecture graphique à l'aide des chemins on trouve comme liste triée de codages de Lebesgue pour S_1 :

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

5 Calcul efficace de l'intersection d'ensembles de points

12. L'AQL de l'ensemble S_1 est : $[[0,0], [0,3], [1,2], [3,4]]$

```
13. def ksuffixe(n,k,q):
    i=n-1
    while i>=n-k and q[i]==4:
        i-=1
    if i==n-k-1:
        nq=list(q)
        nq[i]=4
        return nq
    else:
        return q
```

14. La liste initiale des codages de Lebesgue des points est strictement croissante pour l'ordre lexicographique.

L'invariant de boucle indexée par k que nous maintenons est :

« la liste **temp** est une liste de quadrants strictement croissante pour l'ordre lexicographique représentant le même ensemble de points ».

Lors du parcours de la liste (boucle indexée par i), nous parcourons la liste **temp** pour mettre le compactage dans la liste **res**. (remplacement éventuel par un quadrant complet de côté 2^{k+1}).

```
def compacte(n,s):
    res=list(s)
    for k in range(n):
        temp=list(res)
        res=[]
        p=len(temp)
        i=0
        while i+3<p:
            bloc=ksuffixe(n,k,temp[i])
            if EGAL(bloc,ksuffixe(n,k,temp[i+3])):
                res.append(bloc)
                i+=4
            else:
                res.append(temp[i])
                i+=1
        for j in range(i,p):
            res.append(temp[j])
    return res
```

```
15. def compare_ccodes(n,p,q):
    i=0
    while i<n and p[i]==q[i]:
        i+=1
    if i==n:
        return 0
    elif q[i]==4: # Q a un plus grand quadrant au sens de l'inclusion
        return 2
    elif p[i]==4: # P a un plus grand quadrant au sens de l'inclusion
        return -2
    elif p[i]<q[i]: # P et Q représentent des quadrants de côtés  $2^{n-i-1}$ ...
        return 1 # ... inclus dans un même quadrant de côté  $2^{n-i}$ 
    else: # on a  $p[i]>q[i]$  dans ce cas
        return -1

16. def intersection(n,p,q):
    np=len(p)
    nq=len(q)
    i=0
    j=0
    res=[]
    while i<np and j<nq:
        test=compare_ccodes(n,p[i],q[j])
        if test==0: # égalité des quadrants
            res.append(p[i])
            i+=1
            j+=1
        elif test ==1:
            i+=1
        elif test==-1:
            j+=1
        elif test==2: # on est dans la quadrant q[j]
            res.append(p[i])
            i+=1
        else: # on est dans la quadrant p[i]
            res.append(q[j])
            j+=1
    # une des deux listes a été complètement parcourues
    # les quadrants restants sont hors intersection
    return res
```