Devoir Surveillé #1 MP2I ITC (avec solutions)

Mercredi 2026-04-01 ; durée : deux heures

Sujet extrait de l'épreuve d'informatique B XELCR 2017 : filière MP hors spécialité info, PC, PSI
Les seules différences avec le sujet original sont : la suppression d'une courte seconde partie en SQL ; quelques reformulations mineures visant à clarifier certaines questions.

Remarques préliminaires

Dans toute question, il est possible d'admettre un résultat d'une question précédente. Notamment, il est possible d'utiliser une fonction dont l'écriture a été demandée, même si la question correspondante n'a pas été traitée.

Complexité

La complexité, ou le temps d'exécution d'un programme P est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc.) nécessaires à l'exécution de P. Lorsque cette complexité dépend de plusieurs paramètres n et m, on dira que P a une complexité $O(\phi(m,n))$, lorsqu'il existe trois constantes A, n_0, et m_0 telles que la complexité de P soit inférieure ou égale à $A \times \phi(m,n)$, pour tout $n > n_0$ et $m > m_0$.
Lorsqu'il est demandé de garantir une certaine complexité, le candidat devra justifier cette dernière en raisonnant sur le structure du code.

Rappels de Python

Si a est une liste alors `a[i]` désigne le i-ème élément de cette liste où l'entier i est supérieur ou égal à 0 et strictement plus petit que la longueur `len(a)` de la liste.
La commande `a[i] = e` affecte la valeur de l'expression e au i-ème élément de la liste a. L'expression `[]` construit une liste vide. L'expression `[k]*n` construit une liste de longueur n contenant n occurrences de k. La commande `a = list(b)` construit une copie de la liste b et l'affecte à la variable a. La commande `a.append(x)` modifie la liste a en lui rajoutant un nouvel élément final contenant la valeur de x.

Important : seules les opérations sur les listes apparaissant dans ce paragraphe sont autorisées dans les réponses. Si une fonction Python standard est nécessaire, elle devra être réécrite.

Nous attacherons la plus grande importance à la lisibilité du code produit pas les candidats ; aussi, nous encourageons les candidats à utiliser des commentaires et à introduire des fonctions intermédiaires pour faciliter la compréhension du code.

Intersection de deux ensembles de points

Soit n un entier naturel, on note D_n l'ensemble des entiers naturels compris entre 0 et $2^n - 1$. On appelle « point de $D_n \times D_n$ » tout couple d'entiers $(x, y) \in D_n \times D_n$. Soient P et Q deux parties de $D_n \times D_n$, on cherche à calculer efficacement l'intersection des ensembles de points P et Q. La résolution de ce problème a des applications en simulation numérique, en robotique ou encore dans l'implémentation d'interfaces utilisateurs.
Ce sujet est découpé en quatre parties : la partie **1** porte sur une solution naïve et les parties **2**, **3** et **4** conduisent à la réalisation d'une solution efficace. La partie **1** est indépendante des parties suivantes, qui ne sont pas indépendantes entre elles.

1. Une solution naïve en Python

Pour commencer, un point de coordonnées $(x, y) \in D_n \times D_n$ est représenté en Python par une liste de deux entiers naturels `[x,y]`.
Un ensemble de points est représenté par une liste de points sans répétition, donc comme une liste de listes d'entiers naturels de longueur 2.

1. Écrire une fonction `membre(p,q)` qui renvoie `True` si le point p est dans l'ensemble représenté par la liste q et qui renvoie `False` dans le cas contraire.

   On peut comprendre d'après le sujet que le test d'égalité entre listes n'est pas une opération disponible ; dans ce cas il suffit de faire :

   ```
   def membre(p, q):
       for p2 in q:
           if p[0] == p2[0] and p[1] == p2[1]:
               return True
       return False
   ```

   sinon un simple « `p == p2` » aurait suffit.

2. Écrire une fonction `intersection(p,q)` qui renvoie une liste représentant l'intersection des ensembles représentés par p et q. On implémentera l'algorithme qui consiste à itérer sur tous les points de p et à insérer dans le résultat ceux qui sont aussi dans q

   Le plus simple est de réutiliser la fonction qui vient d'être écrite. On pourrait aussi s'en passer, mais ce serait une perte de temps…

   ```
   def intersection(p, q):
       r = []
       for p2 in p:
           if membre(p2, q):
               r.append(p2)
       return r
   ```

3. Si la comparaison entre deux entiers naturels est prise comme opération élémentaire, quelle est la complexité de l'algorithme de la question précédente exprimée en fonction de la longueur de p et q ?

   On note ℓ_p et ℓ_q les longueurs de p et q respectivement. Si une comparaison est comptée comme opération élémentaire, la fonction *membre* effectue $O(\ell_q)$ opérations élémentaires dans le pire cas. La fonction *intersection* fait ℓ_p appels à *membre*, et $O(\ell_p)$ opérations élémentaires supplémentaires. La complexité de *intersection* est donc un $O(\ell_p \ell_q)$.

2. Codage de Lebesgue

On souhaite implémenter en Python une solution efficace au problème de calcul de l'intersection entre deux ensembles de points. La solution proposée s'appuie sur une structure de données appelée AQL. Cette structure de données suppose que les coordonnées des points sont représentées par leur *codage de Lebesgue*.

Le *codage de Lebesgue* d'un point de coordonnées $(x, y) \in D_n \times D_n$ s'obtient par entrelacement des n bits des représentations binaires sur n bits de x et y en commençant pas les bits de x. On suppose que les bits de poids fort sont situés à gauche dans les représentations binaires des entiers naturels.

Par exemple, si $n = 3$, si x vaut 6 (donc $\overline{\mathbf{110}}^2$ en représentation binaire sur 3 bits) et si y vaut 3 (donc $\overline{011}^2$ en représentation binaire sur 3 bits) alors le codage de Lebesgue du point de coordonnées (x, y) est $\overline{\mathbf{101101}}^2$, c'est à dire la représentation binaire sur 6 bits de 45.

Le codage de Lebesgue d'un point peut être vu comme un nombre écrit dans une base formée des chiffres $\overline{00}^2$, $\overline{01}^2$, $\overline{10}^2$ et $\overline{11}^2$. Ainsi, si $n = 3$, le point $(6, 3) = (\overline{\mathbf{110}}^2, \overline{011}^2)$ est codé par le nombre $\overline{\mathbf{10}}^2\overline{\mathbf{11}}^2\overline{\mathbf{01}}^2$.

De plus, on utilisera la notation décimale 0, 1, 2 et 3 pour représenter les chiffres $\overline{00}^2$, $\overline{01}^2$, $\overline{10}^2$ et $\overline{11}^2$, et l'on notera $\overline{c_{n-1} \ldots c_0}^\ell$ la représentation en base 4 du codage de Lebesgue d'un point de $D_n \times D_n$. Par exemple, pour $n = 3$, le codage de Lebesgue du point $(6, 3)$ sera écrit $\overline{231}^\ell$.

En Python, la séquence des chiffres d'un codage de Lebesgue d'un point de $D_n \times D_n$ est stockée dans une liste de longueur n triées par poids décroissants : le chiffre de poids le plus fort se trouve en première position, le chiffre de poids le plus faible en dernière position. Ainsi, si $n = 3$, le codage de Lebesgue du point $(6, 3)$ est représenté en Python par la liste `[2,3,1]`.

4. Soit $n = 3$, quelle liste Python représente le codage de $(1,6)$?

```
[1,1,2]
```

On suppose que l'on dispose d'une fonctions `bits(x,k)`, qui prend en arguments deux entiers naturels x et k, et qui renvoie la valeur du bit de coefficient 2^k dans la représentation binaire de x.

5. Écrire une fonction `code(n,p)` qui prend en arguments un entier strictement positif n et un point p représenté par une liste de longueur 2 dont les deux coordonnées sont prises dans D_n. Cette fonction renvoie le codage de Lebesgue de p représenté sous la forme d'une liste Python.

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

3. Représentation d'un ensemble de points

On utilise l'ordre lexicographique (autrement dit, l'ordre du dictionnaire) pour trier les codages. Soient $c = \overline{c_{n-1}\dots c_0}^{\ell}$ et $d = \overline{d_{n-1}\dots d_0}^{\ell}$ deux codages de Lebesgue de points de $D_n \times D_n$. On note $c < d$ pour « c est strictement plus petit que d » si :

$$\exists i,\ 0 \leq i < n \ \text{ tel que } \begin{cases} \forall j,\ j > i \Rightarrow c_j = d_j \\ c_i <_{\mathbb{N}} d_i \end{cases}$$

où $<_{\mathbb{N}}$ est l'ordre usuel des entiers naturels.

6. Trier les codages suivants par ordre croissant pour l'ordre lexicographique :

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

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

7. Écrire une fonction `compare_pcodes(n,c1,c2)`, qui prend en arguments deux codages de Lebesgue de $D_n \times D_n$ et renvoie `0` s'il sont égaux, `1` si c_2 est plus grand par l'ordre lexicographique que c_1 et `-1` sinon.

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

Nous allons maintenant représenter un ensemble P de points de $D_n \times D_n$ sous la forme d'une liste triée pour l'ordre lexicographique des codages de Lebesgue des points de P. En guise d'exemple, nous allons coder les points $S_0 = \{(0,0),(1,0),(1,1),(2,2),(3,0),(0,1)\}$ de $D_2 \times D_2$. Ces points sont représentés en noir dans la Fig. 1 ci-dessous, dont l'origine est en bas à gauche, les abscisses croissent de gauche à droite et les ordonnées du bas vers le haut.
On considère d'abord les représentations binaires des coordonnées des points de S_0 :

$$\overline{S_0}^2 = \{(\overline{00}^2, \overline{00}^2), (\overline{01}^2, \overline{00}^2), (\overline{01}^2, \overline{01}^2), (\overline{10}^2, \overline{10}^2), (\overline{11}^2, \overline{00}^2), (\overline{00}^2, \overline{01}^2)\}$$

à partir desquelles on calcule le codage de Lebesgue de chaque point

$$\overline{S_0}^{\ell} = \{\overline{00}^2\overline{00}^2, \overline{00}^2\overline{10}^2, \overline{00}^2\overline{11}^2, \overline{11}^2\overline{00}^2, \overline{10}^2\overline{10}^2, \overline{00}^2\overline{01}^2\} = \{\overline{00}^{\ell}, \overline{02}^{\ell}, \overline{03}^{\ell}, \overline{30}^{\ell}, \overline{22}^{\ell}, \overline{01}^{\ell}\}$$

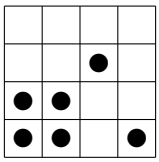


FIGURE 1 – L'ensemble des points S_0

Cet ensemble, une fois trié pour l'ordre lexicographique, s'écrit

$$\{\overline{00}^{\ell}, \overline{01}^{\ell}, \overline{02}^{\ell}, \overline{03}^{\ell}, \overline{22}^{\ell}, \overline{30}^{\ell}\}$$

ce que l'on représente en Python par la liste :

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

Remarquons que nous venons d'effectuer un changement de système de coordonnées. Le codage de Lebesgue du point (x, y) représente le chemin à emprunter pour atteindre (x, y) dans l'espace récursivement divisé en quadrants. En effet, en suivant la numérotation des quadrants donnée dans la Fig. 2 ci-dessous, on s'aperçoit par exemple que le point de coordonnées $(1, 1)$ dans le système de coordonnées usuel est situé dans le quadrant 0 de $D_2 \times D_2$ et qu'à l'intérieur de ce quadrant subdivisé à son tour, le point $(1, 1)$ est dans le quadrant 3.

1	3
0	2

FIGURE 2 – Numérotation des quadrants

On peut vérifier que ces coordonnées $[0, 3]$ dans ce nouveau système correspondent bien au codage de Lebesgue du point de coordonnées $(1, 1)$ du système usuel.

Formellement, pour $k < n$, le quadrant atteint dans $D_n \times D_n$ par le chemin $c = d_1 \cdot d_2 \cdots d_k$ (avec $d_i \in \{0, 1, 2, 3\}$) est défini comme suit :

- Si k vaut 0 alors le chemin c est vide et le quadrant atteint dans $D_n \times D_n$ par c est l'ensemble des points de $D_n \times D_n$.
- Si $k > 0$ alors le chemin est de la forme $d_1 \cdot d_2 \cdots d_k$. Dans ce cas, le quadrant atteint par $d_1 \cdot d_2 \cdots d_k$ dans $D_n \times D_n$ est l'ensemble des points de $D_n \times D_n$ dont le codage de Lebesgue est de la forme $\overline{d_1 d_2 \dots d_k c_{n-k-1} \dots c_0}^{\ell}$ pour $c_{n-k-1}, \dots, c_0 \in \{0, 1, 2, 3\}$.

8. On pose pour cette question $n = 2$. Donner la représentation sous forme de codage de Lebesgue trié par ordre lexicographique de l'ensemble de points :

$$S_1 = \{(0, 0), (3, 3), (3, 2), (1, 1), (1, 2), (2, 2), (2, 3)\}$$

La façon la plus rapide de calculer les codage de Lebesgue des points de S_1 est probablement de faire un dessin et d'utiliser l'interprétation de ce codage donnée juste au dessus par le sujet. On trouve alors :

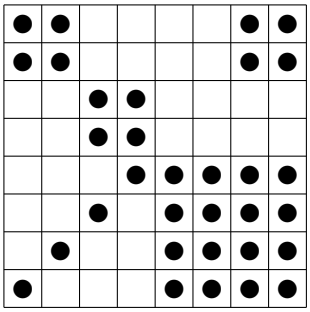
$$\{\overline{00}^{\ell}, \overline{03}^{\ell}, \overline{12}^{\ell}, \overline{30}^{\ell}, \overline{31}^{\ell}, \overline{32}^{\ell}, \overline{33}^{\ell}\}$$

4. Calcul efficace de l'intersection d'ensembles de points

Compaction par codage des quadrants

Soit S un ensemble de points de $D_n \times D_n$ représenté par la liste L triée par ordre lexicographique des codages de Lebesgue de ses points (comme dans la partie précédente). En se dotant d'un symbole supplémentaire, notons le 4, on *compacte* la liste L en représentant chaque sous-séquence (maximale) L' correspondant à un quadrant de chemin $d_1 \cdots d_k$ par l'unique mot $\overline{d_1 \cdots d_k \cdot 4 \cdots 4}^{\ell}$ (dans lequel on a rajouté $n - k$ fois le symbole 4).

Notons qu'un codage de Lebesgue compacté représente *un ensemble de points* et non un unique point comme c'est la cas avec les codages de Lebesgue non compactés. Notons aussi que la liste des codages reste triée par ordre lexicographique.
Par exemple, l'ensemble des points S_0 est compactable en `[[0,4],[2,2],[3,0]]`. Le codage `[0,4]` représente le quadrant 0 situé en bas à gauche de la Fig. 1. Enfin, pour illustrer un cas où $n > 2$, la Fig. 3 décrit le codage compacté d'un ensemble de points de $D_3 \times D_3$.



est compacté en :

$\{\overline{000}^{\ell}, \overline{003}^{\ell}, \overline{030}^{\ell}, \overline{033}^{\ell}, \overline{114}^{\ell}, \overline{124}^{\ell}, \overline{244}^{\ell}, \overline{334}^{\ell}\}$

FIGURE 3 – Un ensemble de points de $D_3 \times D_3$ et sa représentation compactée

Structure de données d'AQL

On appelle « AQL de l'ensemble de points S » la liste triée et compactée des codages de Lebesgue des points de l'ensemble S.

9. Donner l'AQL de l'ensemble S_1 de la question **8**.

Il faut et il suffit de compacter le quadrant *NE*, ce qui donne : `[[0, 0], [0, 3], [1, 2], [3, 4]]`

10. Écrire une fonction `ksuffixe(n,k,q)` qui prend en arguments un entier n strictement positif, une liste q représentant le codage de Lebesgue compacté d'un quadrant de $D_n \times D_n$ et un entier naturel k inférieur strictement à n. Si les k derniers chiffres de la liste q ont pour valeur 4, cette fonction renvoie une nouvelle liste semblable à la liste q mais dont les $k+1$ derniers chiffres valent 4. Sinon, cette fonction renvoie q inchangée.
Ainsi, `ksuffixe(4,2,[0,1,4,4])` renvoie `[0,4,4,4]`, et `ksuffixe(4,2,[0,1,2,4])` renvoie `[0,1,2,4]`.

Il faut bien prendre garde à renvoyer une *nouvelle* liste.

```
def ksuffixe(n, k, q):
    for i in range(n-1,n-k-1,-1):
        if q[i] != 4:
            return q
    q2 = list(q) # attention
    q2[n-k-1] = 4
    return q2
```

11. L'algorithme de compaction d'une liste triée de codages de Lebesgue consiste à parcourir n fois la liste représentant l'ensemble de points. L'itération k vise à remplacer quatre codages successifs formant un quadrant complet de côté 2^{k+1} par la représentation compactée de ce quadrant.
Écrire une fonction `compacte(n,s)` qui prend en arguments un entier strictement positif n et un ensemble de points P de $D_n \times D_n$ représenté par la liste triée s des codages de Lebesgue de ses points. Cette fonction renvoie l'AQL de l'ensemble de points P.

On peut introduire une fonction intermédiaire pour effectuer le traitement de la boucle interne.

```
def one_pass(n, s2, k):
    r = []
    # on crée la liste des ksuffixes de s2
```

```
    ks2 = [ksuffixe(n, k, q) for q in s2]
    i = 0
    # on compare les ksuffixes sur une fenêtre
    # glissante de 4
    # en cas d'égalité on ajoute le nouveau ksuffixe
    # au résultat, sinon on ajoute l'élément initial
    while i < len(ks2) - 3:
        if ks2[i] == ks2[i+1] and ks2[i] == ks2[i+2] and ks2[i] == ks2[i+3]:
            r.append(ks2[i])
            i = i + 4
        else:
            r.append(s2[i]) # attention
            i = i + 1
    # on complète avec la fin de la liste, s'il y en a une
    while i < len(s2):
        r.append(s2[i]) # attention
        i = i + 1
    return r

def compacte(n, s):
    for k in range(n):
        s = one_pass(n, s, k)
    return s
```

12. On remarque que l'ordre lexicographique $<$ défini plus haut s'adapte sans changement aux codages de Lebesgue compactés. Cependant, on souhaite comparer deux codages de Lebesgue compactés en terme de relations d'inclusion et d'exclusion des ensembles de points qu'ils représentent.
Écrire une fonction `compare_ccodes(n,p,q)` qui prend en arguments un entier strictement positif n, une liste p contenant le codage de Lebesgue compacté d'un quadrant P de $D_n \times D_n$ et une liste q contenant le codage de Lebesgue compacté d'un quadrant Q de $D_n \times D_n$. Cinq valeurs de retour sont possibles :

- l'entier `0` si les quadrants sont égaux ;
- l'entier `1` si les quadrants P et Q sont disjoints et $p < q$;
- l'entier `-1` si les quadrants P et Q sont disjoints et $q < p$;
- l'entier `2` si $P \subset Q$;
- l'entier `-2` si $Q \subset P$

Par exemple :

- `compare_ccodes(3,[1,4,4],[2,4,4])` renvoie `1` ;
- `compare_ccodes(3,[1,2,4],[1,4,4])` renvoie `2`.

```
def compare_ccodes(n, p, q):
    for i in range(n):
        if p[i] != q[i]:
            if p[i] == 4:
                return -2
            if q[i] == 4:
                return 2
            if q[i] < p[i]:
                return -1
            return 1
    return 0
```

13. Pour calculer efficacement l'intersection de deux ensembles de points représentés par leur AQL respectif, on *fusionne* les deux listes triées qui leur correspondent.
En utilisant `compare_ccodes`, écrire une fonction `intersection(n,p,q)` qui prend en argument un entier strictement positif n ainsi que deux AQL P et Q représentant respectivement deux ensembles de points de $D_n \times D_n$. Cette fonction renvoie un AQL représentant $P \cap Q$. Le nombre d'appels à la fonction `compare_ccodes` effectués par `intersection(n,p,q)` doit être un $O(\texttt{len}(p) + \texttt{len}(q))$.

On applique l'algorithme de fusion du tri fusion, modifié pour prendre en compte les inclusions.
Plus précisément :

— en cas d'égalité entre `p[ip]` et `q[iq]`, l'ensemble de point correspondant est présent dans l'intersection $P \cap Q$ et est donc ajouté au résultat ; on passe aux éléments suivants de p et q en incrémentant i_p et i_q
— en cas d'inégalité et de non inclusion, on incrémente comme dans un tri fusion le compteur correspondant à l'ensemble de point le plus petit, et on n'ajoute rien au résultat
— on fait de même en cas d'inégalité et d'inclusion, mais cette fois en ajoutant l'ensemble de point inclus dans l'autre au résultat

Le sujet demandant de renvoyer un AQL, on peut (dans une logique de concours) simplement conclure par un appel à *compacte*, plutôt que de justifier qu'on obtient un AQL par construction.

```
def intersection(n, p, q):
    ip = 0
    iq = 0
    r = []
    while ip < len(p) and iq < len(q):
        cmp = compare_ccodes(n, p[ip], q[iq])
        if cmp == 0:
            r.append(p[ip])
            ip = ip + 1
            iq = iq + 1
        elif cmp == 1:
            ip = ip + 1
        elif cmp == -1:
            iq = iq + 1
        elif cmp == 2:
            r.append(p[ip])
            # on avance seulement ip :
            # par ex., deux passes sur p et une
            # sur q nécessaires pour
            # traiter :
            #  x | ·    x | x
            #  · | x    x | x
            ip = ip + 1
        elif cmp == -2:
            r.append(q[iq])
            iq = iq + 1
    # r triée par ordre lexicographique
    r = compacte(n, r) # pas vraiment nécessaire
    return r
```

Comme demandé par le sujet, il faut justifier que la complexité obtenue est bien celle désirée. Une itération de la boucle `while` effectue un seul appel à *compare_ccodes*, et la condition de sortie de boucle ainsi que les mise à jour des compteurs i_p et i_q garantissent que le nombre d'appels à cette fonction est bien un $O(\texttt{len}(p) + \texttt{len}(q))$: la quantité $i_p + i_q$ augmente strictement à chaque itération, et est majorée par $\texttt{len}(p) + \texttt{len}(q)$.

Fin du sujet