Concours Polytechnique filières PSI, PC, MP : corrigé

Jean-Loup Carré

Informatique commune – 2017

Question 1.
```
def membre(p, q):
    for x in q:
        if x == p:
            return True
    return False
```

Question 2.
```
def intersection(p, q):
    r = []
    for x in p:
        if membre(x, q):
            r.append(x)
    return r
```

Question 3. Notons n_p et n_q les longueurs respectives des listes p et q.
Chaque appel à `membre(x, q)` va coûter au plus n_q comparaisons. Pour chaque élément de `p`, on fait une itération du `for`, et donc un appel à `membre(x, q)`.
La complexité en nombre de comparaisons est donc, au pire, $n_p \times n_q$.

Attention ! Piège !

Le sujet demande la complexité en ne comptant qu'une seule opération élémentaire : la comparaison. Ce n'est donc pas la complexité au sens *usuel* qui est demandée.

Cette complexité en nombre de comparaisons peut, dans l'absolu, être différente de la complexité usuelle. Par exemple, dans l'algorithme de tri par insertion, si la position de l'élément à insérer est recherchée par dichotomie, la complexité en nombre de comparaisons ne sera qu'en $\mathcal{O}(n\ln(n))$, ce qui est nettement mieux que la complexité au sens usuel (en $\mathcal{O}(n^2)$) du tri par insertion.

Dans le cas présent, la complexité en nombre de comparaisons et la complexité usuelle ont le même ordre de grandeur.

Question 4. Pour distinguer les paramètres a et b des noms de colonnes, nous les préfixons par un `@`.

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

Remarque

Le statut de a et b n'est pas clair dans la question. Le sujet demanderait-il d'écrire une procédure (ce qui déborde du programme de CPGE qui est assez modeste sur les bases de données) ?
Dans le doute, nous introduisons la notation `@` utilisée par certains dialectes de SQL pour passer des paramètres.

Question 5.
```
SELECT x, y
  FROM POINTS
  JOIN MEMBRE AS M1 ON id = M1.idpoint
  JOIN MEMBRE AS M2 ON id = M2.idpoint
  WHERE M1.idensemble = @i AND M2.idensemble = @j;
```

Remarque

Même remarque qu'à la question précédente.

Question 6.
```
SELECT M1.idpoint
  FROM MEMBRE as M1
  JOIN MEMBRE as M2 ON M1.idensemble = M2.idensemble
  JOIN POINTS ON POINTS.id = M2.idpoint
  WHERE x = @a AND y = @b;
```

Question 7. 1 s'écrit en binaire $\overline{001}^2$ et 6 s'écrit $\overline{110}^2$. Ainsi, le code de Lebesgue du point $(1, 6)$ est $\overline{01}^2, \overline{01}^2, \overline{10}^2$. La liste en Python sera donc `[1, 1, 2]`.

Question 8.
```
def code(n, p):
    c = []
    x, y = p
    for k in range(n-1, -1, -1):
        c.append(2*bit(x, k)+bit(y,k))
    return c
```

Remarque

Le sujet suppose que la fonction `bit` est déjà définie. Elle peut être définie comme suit :

```
def bit(x, k):
    return (x >> k) % 2
```

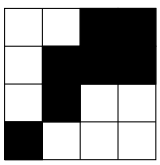
Si on ne connait pas l'opérateur `>>` qui supprime les `k` derniers bits, on peut à la place utiliser la division euclidienne :

```
def bit(x, k):
    return (x // 2**k) % 2
```

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

Question 10.
```
def compare_pcodes(n, c1, c2):
    for k in range(n):
        if c1[k]<c2[k]:
            return 1
        elif c1[k]>c2[k]:
            return -1
    return 0
```

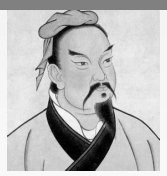
Question 11. Remarque. Cette question demande des éléments de la partie suivante (pour le compactage).
On représente l'ensemble des points en noir.



On ne peut regrouper que les 4 points du quadrant 3.
La liste demandée est finalement : `[[0, 0], [0, 3], [1, 2], [3, 4]]`.

Question 12. Cette question est identique à la précédente.

Conseil stratégique


Sun Tsu

Que deux questions identiques soient posées est très probablement dû à une erreur d'énoncé.
Plutôt que de jouer à Mme Irma et tenter de deviner quelle question aurait voulu poser le concepteur du sujet, on répond à la question telle qu'elle est posée.

Remarque

On fera attention à l'orthographe du mot « **quadrant** » et à ne pas le confondre avec son homophone « cadran ».

Question 13.
```
def ksuffixe(n, k, q):
    for i in range(n-1, n-k-1, -1): #Vérifions que les derniers éléments sont des 4
        if q[i] != 4:
            return q
    q2 = list(q)
    q2[len(q2)-k-1] = 4
    return q2
```

Remarque

L'instruction `q2[len(q2)-k-1] = 4` pourrait être remplacée par `q2[-k-1] = 4`.
Cependant, le sujet demande de n'utiliser que des indices de liste positifs.

Question 14. On suit les instructions de l'énoncé, et on crée une fonction auxiliaire faisant un des parcours demandés.

```
def parcours(n, k, s):
    # On précalcule la liste des suffixes
    su = [ksuffixe(n, k, s[i]) for i in range(len(s))]
    co = []
    i = 0
    while i < len(s):
        # Si on peut compacter les 4 éléments suivants
        if i<len(s)-3 and su[i] == su[i+1] == su[i+2] == su[i+3]:
            co.append(su[i])
            i += 4
        else:
            co.append(s[i])
            i+=1
    return co
```

Il ne reste plus qu'à itérer la fonction précédente.

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

Question 15.
```
def compare_ccodes(n, p, q):
    for k in range(n):
        if p[k]<q[k]:
```

```
            if q[k] == 4:
                return 2
            else:
                return 1
        elif p[k]>q[k]:
            if p[k] == 4:
                return -2
            else:
                return -1
    return 0
```

Question 16. Nous nous inspirons du principe de fusion de listes de l'algorithme de tri par fusion. Nous créons deux indices, i_p et i_q qui vont parcourir pour l'un la liste p, pour l'autre la liste q.

À chaque itération de la boucle de l'algorithme nous comparons P (le $i_p$ème élément de p) à Q (le $i_q$ème élément de q). Nous distinguons alors 4 cas, selon la valeur de `compare_ccodes(n, p[ip], q[iq])`.

```
def intersection2(n, p, q):
    ip = 0
    iq = 0
    r = [] # Liste des éléments à garder
    while ip < len(p) and iq < len(q):
        a = compare_ccodes(n, p[ip], q[iq])
        if a == 2 or a == 0:
            r.append(p[ip])
            ip += 1
        elif a == -2:
            r.append(q[iq])
            iq += 1
        elif a == 1:
            ip += 1
        else :  # Cas a==-1
            iq +=1
    return r
```

Si $P \subseteq Q$, alors on garde P, et on passe à l'élément suivant dans la liste p (on procède symétriquement si $Q \subsetneq P$).
Si P et Q sont disjoints, on élimine l'élément le plus petit entre P et Q en augmentant le compteur idoine.
L'algorithme est correct, car, tout au long de l'algorithme, la quantité $r \cup (p[i_p :] \cap q[i_q :])$ est constante ; et, qu'au début elle vaut évidemment la valeur recherchée $(p \cap q)$, et qu'à la fin elle vaut bien la valeur renvoyée (i.e., r).

Remarque

On a arbitrairement choisi de regrouper le cas `a == 0` avec le cas `a == 2`, on aurait aussi pu le regrouper avec le cas `a == -2`.

Remarque

On utilise un invariant de boucle non-booléen, à savoir $r \cup (p[i_p :] \cap q[i_q :])$. Si on tient absolument à avoir un invariant booléen (i.e., valant *vrai* ou *faux*), on peut prendre à la place l'invariant $r \cup (p[i_p :] \cap q[i_q :]) = p \cap q$.