ENSAE mardi 23 octobre 2012

Cette interrogation écrite compte pour 5 points ajoutés à l'ensemble des notes de la matière. Tous les documents sont autorisés. La durée est d'une demi-heure. Vous devrez imprimer le résultat en n'omettant pas d'y ajouter votre nom et le numéro de l'énoncé qui vous aura été distribué.

1

1) Ecrire une fonction qui retourne la fréquence de chaque lettre d'un mot. Le résultat sera un dictionnaire dont les clés seront les lettres et les valeurs seront les fréquences. La fréquence désigne le nombre d'occurence d'une lettre. (2 points)

2) Ecrire une fonction qui vérifie que deux mots sont des anagrammes : deux mots sont anagrammes l'un de l'autre s'ils sont composés des mêmes lettres. Cette fonction devra inclure au moins une boucle. (3 points)

Correction

```
# coding: latin-1
# question 1
def frequence_lettre (mot) :
    res = { }
    for c in mot :
        if c in res : res[c] += 1
        else : res [c] = 1
    return res

print frequence_lettre ("aviateur")
# affiche {'a': 2, 'e': 1, 'i': 1, 'r': 1, 'u': 1, 't': 1, 'v': 1}

# Deux autres écritures de la fonction
def frequence_lettre (mot) :
    res = { c:0 for c in mot }
    for c in mot : res[c] += 1
    return res

def frequence_lettre (mot) :
    res = { }
    for c in mot :
        # la méthode get retourne res[c] si cette valeur existe, 0 sinon
        res[c] = res.get( c, 0 ) + 1
    return res

# question 2

def anagramme (mot1, mot2) :
    h1 = frequence_lettre(mot1)
    h2 = frequence_lettre(mot2)

    # il fallait éviter d'écrire la ligne suivante bien qu'elle retourne le résultat cherché :
    # return h1 == h2

    for c in h1 :
        if c not in h2 or h1[c] != h2[c] : return False
    # il ne faut pas oublier cette seconde partie
```

```
    for c in h2 :
        if c not in h1 : return False
    return True

a,b = "anagramme", "agrammane"
print anagramme (a,b), anagramme (b,a) # affiche True, True

# on vérifie que la fonctionne marche aussi dans l'autre cas
a,b = "anagramme", "agrummane"
print anagramme (a,b), anagramme (b,a) # affiche False, False

# on pouvait faire plus rapide en éliminant les cas évidents
def anagramme (mot1, mot2) :
    if len(mot1) != len(mot2) : return False
    h1 = frequence_lettre(mot1)
    h2 = frequence_lettre(mot2)
    if len(h1) != len(h2) : return False

    for c in h1 :
        if h1[c] != h2.get(c, 0) : return False
    return True
```

2

1) Ecrire une fonction qui calcule la factorielle de n : $f(n) = n! = 2 * 3 * ... * n$. Le calcul ne doit pas être récursif. (2 points)

2) Ecrire une fonction qui $f(a,b) = f(a-1,b) + f(a,b-1)$ pour $a, b \geqslant 1$. On suppose que $f(0,n) = f(n,0) = n$. Le calcul ne doit pas être récursif. (3 points)

Correction

```
# coding: latin-1

# question 1

def factorielle (n) :
    res = 1
    while n > 1 :
        res *= n
        n -= 1
    return res

print factorielle (3)

# voici la version récursive qui n'était pas demandée :
def factorielle_recursive (n) :
    return n*factorielle_recursive (n-1) if n > 1 else 1

# question 2

def f(a,b) :

    # f(a,b) = f(a-1,b) + f(a,b-1)
    # la formule implique de calculer toutes les valeurs f(i,j)
    # avec (i,j) dans [[0,a]] x [[0,b]]
```

```
    # un moyen afin d'éviter de trop nombreux calculs est de
    # stocker les valeurs intermédiaires de f dans une matrice
    # il faut ensuite s'assurer que f(a-1,b) et f(a,b-1)
    # ont déjà été calculées avant de calculer f(a,b)
    # pour cela, on parcourt la matrice dans le sens des i et j croissants
    # il ne faut pas oublier les a+1,b+1 car range (a) est égal à [ 0,1, ..., a-1 ]

    mat = [ [ 0 for i in range (b+1) ] for j in range (a+1) ]
    for i in range (a+1) :
        for j in range (b+1) :
                if i == 0 or j == 0 :
                    mat [i][j] = max (i,j)
                else :
                    mat [i][j] = mat [i-1][j] + mat[i][j-1]
    return mat [a][b]

# on teste pour des valeurs simples
print f(0,5) # affiche 5
print f(1,1) # affiche 2
# on vérifie en suite que cela marche pour a < b et b > a
print f (4,5)  # affiche 210
print f (5,4)  # affiche 210

# autres variantes

# la version récursive ci-dessous est juste beaucoup plus longue, elle appelle
# la fonction f_recursive autant de fois que la valeur retournée
# par la fonction cout_recursive alors que la fonction précédente
# effectue de l'ordre de O(a*b) calculs
def f_recursive(a,b) :
    return f_recursive(a-1,b) + f_recursive(a,b-1) if a > 0 and b > 0 else max(a,b)

def cout_recursive(a,b) :
    return cout_recursive(a-1,b) + cout_recursive(a,b-1) if a > 0 and b > 0 else 1

# une autre version non récursive
def f(a,b) :
    mat = { }
    for i in range (a+1) :
        for j in range (b+1) :
                mat [i,j] = mat [i-1,j] + mat[i,j-1] if i > 0 and j > 0 else max (i,j)
    return mat [a,b]
```

La fonction `cout_recursive` retourne le nombre d'appels à la fonction `f_recursive` fait pour calculer $f(a,b)$. Pour avoir une idée plus précise de cette valeur, on cherche à déterminer la valeur de la fonction `cout_recursive` en fonction de (a,b) et non de façon récursive. On note la fonction $g =$ `cout_recursive` :

$$g(a,b) = \begin{cases} g(a-1,b) + g(a,b-1) & \text{si } a > 0 \text{ et } b > 0 \\ 1 & \text{sinon} \end{cases} \quad (1)$$

Cette définition est identique à celle du triangle de Pascal sur les combinaisons. On peut donc en déduire que :

$$g(a,b) = C_{a+b}^{a} \tag{2}$$

La version récursive sous cette forme est simple mais très coûteuse. Même pour des petites valeurs, le calcul devient vite très lent. Cela ne veut pas dire qu'il ne faut pas l'envisager à condition de stocker les valeurs intermédiaires :

```
def f_recursive(a,b, memoire) :
    if (a,b) in memoire : return memoire [a,b]
    else :
        res = f_recursive(a-1,b, memoire) + f_recursive(a,b-1, memoire) \
                    if a > 0 and b > 0 else max(a,b)
        memoire [a,b] = res
        return res
```

3

1) Ecrire une fonction qui retourne la somme des éléments de cette liste entre les positions `i` et `j` exclu sans utiliser la fonction `sum`. Testez la fonction sur une liste de votre choix contenant des nombres positifs et négatifs. (2 points)

2) Utiliser la fonction précédente pour trouver la sous-liste `li[i : j]` dont la somme des éléments est la plus grande. Elle inclura deux boucles. (3 points)

Correction

L'une des réponses était suffisante pour obtenir les trois points.

```
# coding: latin-1

# question 1

def somme_partielle (li, i, j) :
    r = 0
    for a in range (i,j) :
        r += li [a]
    return r

# question 2

def plus_grande_sous_liste (li) :
    meilleur = min(li)               # ligne A
    im,jm = -1,-1
    for i in range (0,len(li)) :
        for j in range (i+1, len(li)+1) :   # ne pas oublier +1 car sinon
                        # le dernier élément n'est jamais pris en compte
            s = somme_partielle(li, i,j)
            if s > meilleur :
                meilleur = s
                im,jm = i,j
    return li [im:jm]
```

```
# si li ne contient que des valeurs positives, la solution est évidemment la liste entière
# c'est pourquoi il faut tester cette fonction avec des valeurs négatives
li = [ 4,-6,7,-1,8,-50,3]
m  = plus_grande_sous_liste(li)
print m   # affiche [7, -1, 8]

li = [1,2,3,4,5,-98,78,9,7,7]
m  = plus_grande_sous_liste(li)
print m   # affiche [79, 9, 7, 7]

# autre version plus courte

def plus_grande_sous_liste (li) :
    solution = [ (somme_partielle(li,i,j), i, j) \
                    for i in range (0,len(li)) \
                    for j in range (i+1, len(li)+1) ]
    m = max(solution)
    return li [m[1]:m[2]]
```

La ligne A contient l'instruction `meilleur = min(li)`. Pour une liste où tous les nombres sont négatifs, la meilleure sous-liste est constitué du plus petit élément de la liste. Remplacer cette instruction par `meilleur = 0` a pour conséquence de retourner une liste vide dans ce cas précis. En ce qui concerne le coût de la fonction, les deux solutions proposées sont équivalentes car elles effectuent une double boucle sur la liste. Le coût de l'algorithme est en $O(n^3)$:

$$cout(n) = \sum_{i=0}^{n} \sum_{j=i+1}^{n} j - i \tag{3}$$

$$= \sum_{i=0}^{n} \sum_{j=0}^{i} j \tag{4}$$

$$= \sum_{i=0}^{n} \frac{i(i+1)}{2} \tag{5}$$

Il est possible de modifier cette fonction de telle sorte que le coût soit en $O(n^2)$ car on évite la répétition de certains calculs lors du calcul de la somme des sous-séquences.

```
def plus_grande_sous_liste_n2 (li) :
    meilleur = 0
    im,jm = -1,-1
    for i in range (0,len(li)) :
        s = 0
        for j in range (i, len(li)) :
            s += li[j]
            if s > meilleur :
                meilleur = s
                im,jm = i,j+1
    return li [im:jm]

li = [ 4,-6,7,-1,8,-50,3]
m  = plus_grande_sous_liste_n2(li)
print m   # affiche [7, -1, 8]
```

```
li = [1,2,3,4,5,-98,78,9,7,7]
m  = plus_grande_sous_liste_n2(li)
print m   # affiche [79, 9, 7, 7]
```

Enfin, il existe une dernière version plus rapide encore. Si on considère la liste $l = (l_1, ..., l_n)$ dont on cherche à extraire la sous-séquence de somme maximale. Supposons que l_a appartienne à cette sous-séquence. On construit la fonction suivante :

$$f(a,k) = \left\{ \begin{array}{ll} \sum_{i=a}^{k} l_i & \text{si } k > a \\ \sum_{i=k}^{a} l_i & \text{si } k < a \end{array} \right. \tag{6}$$

On cherche alors les valeurs k_1 et k_2 telles que :

$$f(a,k_1) = \max_{k<a} f(a,k) \tag{7}$$

$$f(a,k_2) = \max_{k>a} f(a,k) \tag{8}$$

La sous-séquence de somme maximale cherchée est $[k_1, k_2]$ avec a dans cet intervalle et le coût de cette recherche est en $O(n)$. Mais ceci n'est vrai que si on sait que l_a appartient à la sous-séquence de somme maximale.

Autre considération : pour deux listes l_1 et l_2, la séquence maximale de la réunion des deux appartient soit à l'une, soit à l'autre, soit elle inclut le point de jonction.

Ces deux idées mises bout à bout donne l'algorithme suivant construit de façon récursive. On coupe la liste en deux parties de longueur égale :
– On calcule la meilleure séquence sur la première sous-séquence.
– On calcule la meilleure séquence sur la seconde sous-séquence.
– On calcule la meilleure séquence en suppose que l'élément du milieu appartient à cette séquence.
La meilleure séquence est nécessairement l'une des trois.

```
#coding:latin-1
def plus_grande_sous_liste_rapide_r (li, i,j) :
    if i == j : return 0
    elif i+1 == j : return li[i],i,i+1

    milieu = (i+j)/2

    # on coupe le problème deux
    ma,ia,ja = plus_grande_sous_liste_rapide_r (li, i, milieu)
    mb,ib,jb = plus_grande_sous_liste_rapide_r (li, milieu, j)

    # pour aller encore plus vite dans un cas précis
    if ja == ib :
        total = ma+mb
        im,jm = ia,jb
    else :
        # on étudie la jonction
        im,jm = milieu,milieu+1
```

```
        meilleur = li[milieu]
        s = meilleur
        for k in range (milieu+1, j) :
            s += li[k]
            if s > meilleur :
                meilleur = s
                jm = k+1

        total = meilleur
        meilleur = li[milieu]
        s = meilleur
        for k in range (milieu-1, i-1, -1) :
            s += li[k]
            if s > meilleur :
                meilleur = s
                im = k

        total += meilleur - li[milieu]

    if   ma >= max(mb,total) : return ma,ia,ja
    elif mb >= max(ma,total) : return mb,ib,jb
    else : return total,im,jm

def plus_grande_sous_liste_rapide (li) :
    m,i,j = plus_grande_sous_liste_rapide_r (li,0,len(li))
    return li[i:j]

li = [ 4,-6,7,-1,8,-50,3]
m  = plus_grande_sous_liste_rapide(li)
print m   # affiche [7, -1, 8]

li = [1,2,3,4,5,-98,78,9,7,7]
m  = plus_grande_sous_liste_rapide(li)
print m   # affiche [79, 9, 7, 7]
```

Le coût de cette fonction est au pire en $O(n \ln n)$ et c'est presque la réponse optimale à cette seconde question.