Correction du DS 3

Informatique de tronc commun, première année

Julien Reichert

Exercice 1

On utilise ici un dictionnaire, par principe et pour encourager le recours à cette structure surpuissante et étudiée plus en détail en deuxième année.

Une fois le nombre d'occurrences par valeur connu, il reste à trouver un indice réalisant le maximum du dictionnaire.

Si les éléments de `L` ne peuvent pas être des clés de dictionnaire, on peut toujours le faire à la main...

```
def plus_frequent(L):
    dict = {}
    for sousliste in L:
        for element in sousliste:
            if element in dict:
                dict[element] += 1
            else:
                dict[element] = 1
    rep = L[0] # initialisation possible
    for cle in dict:
        if dict[cle] > dict[rep]:
            rep = cle
    return rep # Première valeur parmi celles à égalité depuis Python 3.7
```

Exercice 2

On récupère ici le code de la fonction de fusion issue du tri fusion.

(Version alternative par rapport à la correction du TP 8.)

```
def merge(l1, l2):
    n1, n2, i1, i2 = len(l1), len(l2), 0, 0
    l = []
    while i1 < n1 and i2 < n2:
        if l1[i1] < l2[i2]:
            l.append(l1[i1])
            i1 += 1
        else:
            l.append(l2[i2])
            i2 += 1
    l.extend(l1[i1:]) # vide si i1 vaut n1
    l.extend(l2[i2:]) # vide si i2 vaut n2
    return l
```

Exercice 3

Bien entendu, il faut commencer par récupérer une version triée de l'entrée. Une astuce possible revient à prendre un tri non en place (tri fusion par exemple) et à changer la condition pour la comparaison, d'où la réécriture de la fonction merge (ici la version de la correction du TP 8 adaptée, par souci pédagogique).

Attention à faire un tri décroissant !

```
def merge(l1, l2):
    n1, n2, i1, i2 = len(l1), len(l2), 0, 0
    l = [None] * (n1+n2)
    while i1 < n1 and i2 < n2:
        if l1[i1][1] > l2[i2][1]:
            l[i1+i2] = l1[i1]
            i1 += 1
        else:
            l[i1+i2] = l2[i2]
            i2 += 1
    for i in range(i1, n1): # vide si i1 = n1, termine l
            l[i+i2] = l1[i]
    for i in range(i2, n2): # vide si i2 = n2, termine l
            l[i1+i] = l2[i]
    return l

def merge_sort(l):
    n = len(l)
    if n <= 1:
        return l
    l1 = merge_sort(l[:n//2])
    l2 = merge_sort(l[n//2:])
    return merge(l1, l2)

def classement(liste):
    ordre = merge_sort(liste)
    place = 1
    valeur_precedente = ordre[0][1]
    reponse = [(1, ordre[0][0])]
    for i in range(1, len(ordre)):
        (nom, valeur) = ordre[i]
        if valeur_precedente != valeur:
            place = i + 1 # On réinitialise la place qui ne peut être que l'indice actuel plus un
            valeur_precedente = valeur
        reponse.append((place, nom))
    return reponse
```

Exercice 4

Il s'agit désormais, si une valeur est inférieure à la précédente, de remonter parmi toutes les égalités au rang précédent. Seule la fonction classement change, et elle est ici réécrite dans sa totalité pour la clarté.

```
def classement(liste):
    ordre = merge_sort(liste)
    place = 1
    valeur_precedente = ordre[0][1]
    reponse = [(1, ordre[0][0])]
    for i in range(1, len(ordre)):
        (nom, valeur) = ordre[i]
        if valeur_precedente != valeur:
            j = i-1 # début changement
            if reponse[j][0] != j+1:
                (debut, fin) = (reponse[j][0], i)
                while j >= 0 and reponse[j][0] == debut:
                    reponse[j] = ((debut, fin), reponse[j][1])
                    j -= 1 # fin changement
            place = i + 1
            valeur_precedente = valeur
        reponse.append((place, nom))
    j = i # début changement, traiter le cas final
    if reponse[j][0] != j+1:
        (debut, fin) = (reponse[j][0], i+1)
        while j >= 0 and reponse[j][0] == debut:
            reponse[j] = ((debut, fin), reponse[j][1])
            j -= 1 # fin changement
    return reponse
```

Exercice 5

Cette fonction réalise un tri par sélection décroissant et non en place, sans copie de la liste d'entrée comme dans les versions qu'on peut voir de manière classique, mais à la place en utilisant un seuil au-delà duquel les valeurs sont ignorées.

Question 6.1

La valeur de `ts` est augmentée de `l[fin]` à chaque fois que la variable `fin` est incrémentée (avant l'incrémentation) et diminuée de `l[debut]` à chaque fois que la variable `debut` est incrémentée (avant l'incrémentation aussi). Dans ce cas, au vu des conditions initiales, la valeur de `ts` est à tout moment la somme des éléments de `l` entre l'indice `debut` inclus et l'indice `fin` exclu.

Question 6.2

Dans ce cas, on ne peut augmenter `debut`, au vu de la condition, que si la somme totale est strictement supérieure à la somme qu'on veut atteindre, ce qui est impossible quand `debut` vaut `fin` puisque la somme qu'on veut atteindre est garantie d'être positive. Ainsi, `debut` sera forcément toujours inférieure ou égale à `fin`.

Question 6.3

De plus, la variable `fin` ne peut être augmentée que si elle est strictement inférieure à la taille de la liste, faute de quoi la fonction s'arrête si la condition qui faisait incrémenter cette variable était remplie. Par conséquent, à chaque tour de boucle, soit on incrémente `debut`, soit on incrémente `fin`, soit la fonction s'arrête, avec une borne supérieure sur les deux variables permettant de proposer comme variant `2 * len(l) - debut - fin`. La fonction termine donc forcément.

Question 6.4

En ce qui concerne la correction, on observe que la somme stockée dans `ts` augmente ou diminue au fur et à mesure de l'évolution de sa comparaison à `s`, mais en restant toujours la somme d'une tranche de la liste. Un sens de l'implication est facile à prouver : si on retourne un couple `(debut, fin)`, de par les considérations précédentes et la condition qui y a mené, c'est forcément que la tranche `l[debut:fin]` est de somme `s`. Quant à la réciproque, on peut l'établir en supposant qu'il existe deux indices `a` et `b` tels que `a <= b` et `l[a:b]` soit égale à `s`, et on considère le plus petit tel couple. Puisque l'évolution des indices `debut` et `fin` amène forcément `fin` à atteindre `b`, au moment où cela se passe l'indice `debut` ne peut jamais être strictement supérieur à `a` car la somme aurait alors été inférieure ou égale à `s` au tour de boucle précédent, interdisant d'augmenter la valeur de `fin`. Dans ce cas, puisque la somme ne sera alors jamais inférieure strictement à `s`, c'est cette fois-ci uniquement `debut` qui sera incrémentée jusqu'à atteindre `a` et détecter le succès.

Question 6.5

Finalement, une autre façon d'écrire le programme en plus court :

```
def sscbis(l, s):
    for i in range(len(l)):
        for j in range(i, len(l)+1):
            if sum(l[i:j]) == s:
                return (i, j)
    return -1
```

Cependant on note ici une complexité en $\mathcal{O}(n^3)$ où n est la taille de la liste, qu'on peut ramener à une complexité quadratique en ajoutant quelques lignes de code (cf. ci-dessous), mais pas à la complexité linéaire du programme de l'énoncé. En contrepartie, les éléments de la liste peuvent cette fois être négatifs.

```
def ssc2(l, s):
    for i in range(len(l)):
        ts = 0
        for j in range(i, len(l)+1):
            if ts == s:
                return (i, j)
            if j < len(l):
                 ts += l[j]
    return -1
```