DM 1 - PT* ; PT ; PSI* - 2015 : Correction

Exercice 1. Le tri par insertion, tel qu'il a été présenté en cours, est dit **en place**. Cela signifie que la fonction travaille directement sur la liste `L` à trier et ne demande donc aucune occupation supplémentaire en mémoire. Bien entendu, en modifiant la liste, l'algorithme perd la trace de la liste initiale.

Écrire une fonction effectuant le tri par insertion et sauvegardant la liste initiale (sans utiliser `deepcopy`). On travaillera sur une liste `Tr` initialement vide, que l'on remplira au fur et à mesure. La fonction prendra comme argument une liste `L` non triée.

Exercice 2. Tri par insertion avec insertion dichotomique

L'algorithme du tri par insertion consiste à insérer un par un les éléments dans une liste triée. On peut garder ce principe en rendant plus efficace l'insertion elle-même.

Dans cet exercice, la complexité sera évaluée en fonction de $n =$`len(L)`.

(1) Écrire une fonction `insert_dicho(L,a)` qui insère à la bonne place l'élément `a` dans une liste déjà triée `L`, non pas en suivant l'algorithme du tri par insertion, mais de manière dichotomique.

(2) Évaluer la complexité de cette fonction.

(3) Quelle est la complexité globale de l'algorithme de tri par insertion dichotomique obtenu en utilisant cette version de l'insertion ?

(4) On dit que la liste L de longueur n est presque triée si $L_i < L_{i+1}$ pour tout $0 \leqslant i \leqslant n-1$, sauf pour un nombre borné p de valeurs de i.
Quelle est la complexité de l'algorithme de tri par insertion avec insertion dichotomique dans le cas d'une liste presque triée ?

Exercice 3. Bits de parité et codes de Hamming (extrait banque PT 2015)

Un signal transmis peut comporter des erreurs dans les données transmises ; il est indispensable de détecter ces erreurs, et, dans la mesure du possible, de les corriger sans nécessiter une nouvelle transmission.

Une technique très simple pour s'assurer qu'une donnée binaire sera correctement lue par son récepteur est de lui adjoindre un bit de parité, égal, par définition, à :

– 0 si la donnée comporte un nombre pair de 1 (et donc si ses bits sont de somme paire) ;

– 1 si la donnée comporte un nombre impair de 1 (et donc si ses bits sont de somme impaire)

Après réception de la donnée, le récepteur recalcule le bit de parité, et le compare à celui que l'émetteur lui a adressé. Si la donnée n'a pas été altérée lors de la transmission, alors les deux bits de parité sont forcément identiques.

(1) Écrire une fonction `parite(bits)` qui prend en argument une liste `bits`, constituée d'entiers valant 0 ou 1, et retournant l'entier 0 ou 1 correspondant à son bit de parité.

Le code de Hamming est un exemple d'utilisation de bits de parité pour détecter et corriger des erreurs. Nous nous intéressons ici au code (7,4), ainsi appelé car il consiste à joindre trois bits de parité à quatre bits de données, ce qui donne un message de longueur totale de sept bits. Ces trois bits de parité sont ainsi définis : si la donnée s'écrit (d_1, d_2, d_3, d_4) avec $d_i = 0$ ou 1 alors :

– p_1 est le bit de parité du triplet (d_1, d_2, d_4)
– p_2 est le bit de parité du triplet (d_1, d_3, d_4)
– p_3 est le bit de parité du triplet (d_2, d_3, d_4)
Le message encodé, que l'on transmet, s'écrit $(p_1, p_2, d_1, p_3, d_2, d_3, d_4)$

(2) Écrire une fonction `encode_hamming(donnees)` qui prend en argument une liste `donnees` de quatre bits (constitués d'entiers 0 ou 1) et qui retourne une liste de bits contenant le message encodé. On utilisera la fonction précédente.

Le contrôle après réception d'un message ainsi encodé est relativement simple. On pourrait naturellement recalculer les trois bits de parité de la donnée et les comparer aux valeurs transmises, mais la technique proposée par Hamming est de calculer les *trois bits de contrôle* suivants, notés (c_1, c_2, c_3) à partir du *message complet* (données et bits supplémentaires) noté $(m_1, ..., m_7)$:

- c_1 est le bit de parité de l'ensemble (m_4, m_5, m_6, m_7)
- c_2 est le bit de parité de l'ensemble (m_2, m_3, m_6, m_7)
- c_3 est le bit de parité de l'ensemble (m_1, m_3, m_5, m_7)

On montre que si le message a été bien encodé selon les règles précédentes et n'a pas été altéré, alors les trois bits de contrôle doivent être nuls ; si ce n'est pas le cas, il y a eu une erreur. L'intérêt de la technique de Hamming est que, dans le cas particulier où l'erreur est unique, le mot de contrôle donne la représentation binaire de la position de cette erreur *en numérotant à partir de 1.* Par exemple, si $(c_1, c_2, c_3) = (0, 1, 1)$ alors l'erreur porte sur le troisième bit du message. Il suffit ainsi d'inverser ce bit (le mettre à 1 si il est à 0 et inversement) pour corriger l'erreur.

La donnée décodée est alors constituée des quatre bits (d_1, d_2, d_3, d_4) qui se trouvent respectivement en positions 3, 5, 6 et 7 (toujours numérotés à partir de 1) conformément à la description de l'encodage donnée ci-dessus.

(3) Écrire une fonction `decode_hamming(message)` qui prend en argument une liste `message` de sept bits (constitués d'entiers 0 ou 1) et retournant une liste de quatre bits contenant la donnée décodée. En cas d'erreur, on affichera à l'écran un avertissement indiquant la position du bit affecté et on effectuera la correction. On supposera dans cette question que s'il y a une erreur, alors elle est unique.

(4) Déterminer le codage de Hamming de la donnée `1011`, puis la donnée décodée par l'algorithme dans l'hypothèse où les deux premiers bits du message codé ont été incorrectement transmis. Quel a été l'effet de la « correction » sur la donnée dans ce cas ?

(5) Sans coder, proposer un moyen simple de différencier une erreur double d'une erreur unique au moyen d'un bit de parité supplémentaire, et expliquer comment cela permet d'éviter le problème mis en évidence à la question précédente. On s'appuiera sur les méthodes introduites dans cette partie. On ne demande pas d'essayer de corriger la double erreur.

Exercice 1.

```
def tri_insertion2(L):
    M = [L[0]]
    for i in range(1,len(L)):
        M.append(L[i])
        j = i
        while j > 0 and M[j-1] > M[j]:
            M[j], M[j-1] = M[j-1], M[j]
            j -= 1
    return M
```

Exercice 2.

1. L'énoncé ne demande que d'écrire la fonction d'insertion, pas le tri en lui-même, qui reprend de toute façon l'algorithme vu en cours.

```
def insert_dicho(L,e): # version iterative avec while
    m = len(L)
    a, b = 0, m-1
    d = b-a
    while d >= 0:
        i = a + (b-a)//2
        if L[i] == e:
            return L[:i]+ [e] + L[i:]
        elif L[i] < e:
            a, d = i+1, b-a
         else:
            b, d = i-1, b-a
    return L[:a] + [e] + L[a:]
```

```
def insert_dicho_rec(L,e): # version récursive
    if L == []:
        return [e]
    i = len(L)//2
    if L[i] == e:
         return L[:i] + [e] + L[i:]
    elif L[i] < e:
         return L[:i+1] +
insert_dicho_rec(L[i+1:],e)
     else:
         return insert_dicho_rec(L[:i],e)+L[i:]
```

2. L'opération d'insertion se fait a priori en $\mathcal{O}(\ln n)$ opérations de comparaison. C'est très proche d'une question de cours de Sup. Pour une idée de démonstration, insérer un élément a revient si l'on se contente de compter le nombre de comparaisons à trouver deux éléments consécutifs $L[i]$ et $L[i+1]$ tels que $L[i] < a < L[i+1]$.

Au départ, en posant conventionnellement $L[-1] = -\infty$ et $L[n] = +\infty$ (ou tout autre élément arbitrairement petit et grand respectivement à l'ordre considéré), on a $L[-1] < a < L[n]$, donc un encadrement de a par un intervalle entier d'indice de longueur n+1. Or chaque nouvelle itération de ces comparaisons divise la taille de l'intervalle des indices par 2. On a ainsi, de manière approximative, avec t_i la taille de l'intervalle des indices d'encadrement après i itérations : $t_0 = n+1$ et $t_{i+1} = \frac{1}{2}t_i$, suite géométrique de raison $\frac{1}{2}$. On a $t_i \approx \frac{n}{2^i}$ et $t_i = 1$ quand $i = \log_2 n$.

Il faut néanmoins garder à l'esprit que ne dénombrer que le nombre de comparaisons pour analyser l'efficacité algorithmique des algorithmes de tri est au mieux une première approximation. Ici, cette première approximation donne une idée fausse de ce qui se passe en mémoire. En effet, le slicing d'une liste n'est pas une opération qui est en $\mathcal{O}(1)$ mais en $\mathcal{O}(k)$ où k est la taille de la liste à slicer. Ces opérations ne sont certes pas comparables, une comparaison étant *a priori* plus longue en terme de temps qu'un accès ou une copie d'élément. Néanmoins, relativement à la copie de listes et aux accès, l'insertion dichotomique est toujours en $\mathcal{O}(n)$ et même en $\mathcal{O}(n^2)$ pour sa version récursive. Ainsi, *si l'on tient compte de ce qui se passe sous le capot*, ce qu'on doit faire en pratique, l'insertion dichotomique ne sera pas plus efficace que l'insertion habituelle.

3. En sommant, il vient une complexité de l'ordre de $\sum_{k=1}^{n} \ln k \sim n \ln n$, l'équivalent se trouvant en utilisant la technique de comparaison série-intégrale :

Par exemple, en posant $f(t) = \ln(t)$, f étant croissante sur $[1, +\infty[$ on a :

$$\int_1^n \ln t \ \mathrm{d}t \leqslant \sum_{k=2}^{k=n} \ln k \leqslant \int_2^{n+1} \ln t \ \mathrm{d}t$$

Une primitive de $\ln t$ étant $t \ln t - t$, il vient :

$$n \ln n \leqslant \sum_{k=1}^{k=n} \ln k \leqslant (n+1)\ln(n+1) - 2\ln(2)$$

puis :

$$1 \leqslant \frac{\sum_{k=1}^{k=n} \ln k}{n \ln n} \leqslant \frac{n+1}{n} \frac{ln(n+1) - 2\ln 2}{\ln n}$$

et par passage à la limite on a :

$$\frac{\sum_{k=1}^{k=n} \ln k}{n \ln n} \sim 1$$

et :

$$\sum_{k=1}^{k=n} \ln k \sim n \ln n$$

4. La variante dichotomique est une très mauvaise idée si la liste est presque triée. En effet, l'élément à insérer sera, sauf dans p cas, à insérer en dernière place, ce qui nécessite une seule comparaison avec l'insertion classique, mais $\ln n$ comparaisons avec la variante dichotomique. Aussi l'algorithme classique est-il de complexité linéaire alors que la variante dichotomique reste en $\mathcal{O}(\ln n)$.

Exercice 3.

1. Calcul d'un bit de parité

```
def parite(bits):
    return sum(bits)%2
```

2. Encodage de Hamming

```
def encode_hamming(donnees):
    d = donnees
    return [parite([d[0],d[1],d[3]]), parite([d[0],d[2],d[3]]), d[0], parite(d[1:])]+d[1:]
```

3. Décodage de Hamming

```
def decode_hamming(message):
    m = message
    c = parite([m[0],m[2],m[4],m[6]]) + 2*parite([m[1],m[2],m[5],m[6]]) + 4*parite(m[3:]))
    if c != 0:
        print("le bit d'indice "+str(c)+" a été mal transmis")
        cor = m.pop(c)
        m.insert(c, 1-cor)
    return [m[2]] + m[4:]
```

4. En tapant dans la console `encode_hamming([1,1,0,1])` on obtient : `[1, 0, 1, 0, 1, 0, 1]`

Pour trouver le message obtenu en décodant un message où les deux premiers bits ont été altérés, on tape dans la console :

`encode_hamming([0, 1, 1, 0, 1, 0, 1])` on obtient :

`le bit d'indice 3 a été mal transmis [1, 1, 0, 1]`

5.

En cas d'une double erreur, le code détecte une anomalie. Cependant le syndrome propose un chiffre binaire correspondant toujours à une troisième colonne différente des deux précédentes. Ainsi une troisième erreur est ajoutée, car le code ne dispose d'aucun moyen pour différencier une simple d'une double erreur. Pour pallier cet état de fait, ainsi que pour obtenir un code de dimension exactement égale à un octet, un huitième bit est souvent ajouté, il correspond à la parité des sept bits précédents. Une deuxième erreur est ainsi détectée, elle ne peut être corrigée, en revanche l'information d'une double erreur est disponible remplaçant une tentative maladroite, par une demande de retransmission.

Le décodage se fait de la manière suivante :

- S'il n'y a aucune erreur le syndrome est nul.
- Si une erreur s'est produite sur les sept premiers bits, les trois premiers éléments du syndrome donnent la position de l'erreur. L'existence d'un nombre d'erreur impair est confirmé par le huitième bit.
- Si deux erreurs se sont produites, les trois premiers éléments du syndrome ne sont pas tous nuls et le huitième bit indique une parité exact, signal d'un nombre pair d'erreurs. Une retransmission est nécessaire.
- Si une erreur s'est produite sur le huitième bit, l'absence d'erreur sur les trois premiers éléments du syndrome permet de localiser l'erreur et le message est validé.