DM 1

Exercice 1. 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 2. 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 à 0. 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.