DM ITC² n°2 (5/2) :

Algorithmique fondamentale - Récursivité - modèle entité-association

À remettre le ———

Note finale : Calcul du nombre d'inversions dans une liste /

Partie 1 : Calcul du nombre d'inversions dans une liste

La majorité des sites internet de rencontre, les réseaux sociaux, l'analyse du "ranking" avec Google, exploitent des algorithmes de détermination du nombre d'inversions dans une liste.
On fait par exemple appel à ce type d'algorithme, lorsqu'il s'agit de réaliser la comparaison des goûts de deux personnes ayant classé des oeuvres musicales par ordre de préférence.

On considère une liste de n entiers dans laquelle on souhaite déterminer le nombre d'inversions.
On donne un exemple de recensement des inversions dans la liste L ci-dessous :

L= | **10** | **7** | **3** | **14** |

↓ **7 passe avant 10 (1 inversion)**

L= | **7** | **10** | **3** | **14** |

↓ **3 passe avant 10 et avant 7 (2 inversions)**

L= | **3** | **7** | **10** | **14** |

↓ **14 ne bouge pas (0 inversions)**

L= | **3** | **7** | **10** | **14** |

Ainsi, on recense au total **3 inversions** dans la liste originelle.

I Algorithme naïf

□ **1-** Compléter l'algorithme naïf `nbinv_naif(L)` renvoyant le nombre d'inversions dans la liste `L` :

Listing 1 – Recensement des inversions version naïve

```
1  def nbinv_naif(L):
2      nbinv=0
3      for ----------:
4          for ----------:
5              ----------
6              ----------
7      return ----------
```

□ **2-** Déterminer la complexité de cet algorithme. Conclure.

II Algorithme récursif performant : «diviser pour régner»

On cherche désormais à abaisser la complexité de l'algorithme précédent en exploitant le paradigme «diviser pour régner».

Le principe de l'algorithme est le suivant :

- La fonction principale sépare la liste en deux sous-listes de taille identique (ou presque) et ce, de manière récursive.
- Ces sous-listes sont ensuite fusionnées **en procédant à leur tri**, **tout en comptant le nombre d'inversions**.
- Attention : pour faciliter son comptage, le nombre d'inversions cumulées au fur et à mesure de la procédure de tri **sera concaténé en fin de liste**

□ 3- Fonction de coupure récursive

On suppose disposer dans cette question de la fonction `fusion_compt(L1,L2)` chargée de réaliser la fusion des sous-listes $L1$ et $L2$ ainsi que le recensement des inversions.

Ecrire une fonction `nbinv_rec(L)` qui réalise récursivement la coupure de la liste en deux sous-listes et commande la fusion de celle-ci. **On n'omettra pas de concaténer en fin de liste L une valeur initialement nulle, permettant d'inclure ensuite le cumul du nombre d'inversions lors des différentes fusions.**

□ 4- Fonction de fusion et comptage

On se propose désormais de rédiger la fonction `fusion_compt`.

Lors de la fusion, il faudra bien faire attention de compter correctement le nombre d'inversions cumulées.

Supposons que nous en soyons à l'étape de fusion des deux sous-listes **triées** suivantes :

3	7	10	14	18	19

2	11	16	17	23	25

On remarque que si l'on insère un élément de la sous-liste de droite dans la liste finale avant un élément de la sous-liste de gauche, cela signifie qu'il y a une inversion.

a· Expliquer alors pourquoi cela entraine automatiquement qu'il y a autant d'inversions supplémentaires à compter qu'il y a d'éléments restants dans la sous-liste de gauche.

b· A partir des sous-listes ci-dessus, compter par exemple le nombre d'inversions à recenser lorsque l'on insère 2 dans la liste finale.

c· En vous appuyant sur la réponse à la question précédente, compléter le code de la fonction `fusion_compt(L1,L2)` chargée de réaliser la fusion des sous-listes triées ainsi que le comptage et l'affichage du nombre total d'inversions. Dans ce code, à chaque étape de tri et fusion, on stockera, comme prévu, le nombre d'inversions cumulées recensées en dernière position de la liste renvoyée :

Listing 2 – Fusion des sous–listes et comptage des inversions

```
1  def fusion_compt(L1,L2):
2      a_ajouter1,a_ajouter2=L1.pop(),L2.pop() #retrait du dernier
       élément de chaque liste (cumul des inversions)
3      a_ajouter=0
4      n1,n2=len(L1),len(L2)
5      auxil=[]
6      i,j=0,0
7      while (i<n1) and (j<n2):
8          if L1[i] > L2[j]:
9              auxil.append(L2[j])
10             ----------
11             ----------
12         else:
13             auxil.append(L1[i])
14             ----------
15     if j!=len(L2):
16         auxil+=L2[j:]
17     else:
18         auxil+=L1[i:]
19     auxil.append(----------)
20     return auxil
```

d· Rédiger enfin la fonction `nb_inv(L)` exploitant `nbinv_rec(L)` qui renvoie le nombre d'inversions présentes dans la liste L.

e· Déterminer enfin par calcul la complexité «en gros» de cet algorithme. On distinguera le meilleur et le pire des cas. Conclure.

NB : On pourra supposer, pour simplifier la rédaction, que le nombre d'éléments de la liste à traiter est une puissance de 2.

Partie 2 : Principe des codes "Checksum" ou codes correcteurs

La somme de contrôle ou "checksum" en anglais, parfois appelée «empreinte», est un nombre ajouté à des données à transmettre (ou bien à stocker sur un support d'archivage), afin de vérifier la parfaite identité entre le message émis et celui reçu (ou bien "déstocké" depuis son support). Le principe est élémentaire et consiste à former selon un algorithme, et à ajouter aux données à transmettre (ou à conserver), un code qui dépend de celles-ci ; on parle alors de *redondance*. Au moment de la réception (ou du déstockage des données), on calculera par le même algorithme cette somme de contrôle à partir des données reçues (ou déstockées) et on le comparera à la somme de contrôle transmise avec les données ; l'égalité des deux sommes de contrôle assure alors l'intégrité de la transmission du message. A contrario, une discordance de la valeur des deux sommes signale qu'il y a eu corruption des données.

La clé d'un code INSEE ou bien le 13ième chiffre d'un code barre en sont deux exemples simples.

L'énoncé qui suit propose d'étudier quelques algorithmes de génération de somme de contrôle.

I Codage par contrôle de parité (ou VRC, pour *Vertical Redundancy Checking*)

Il s'agit d'un des systèmes les plus simples ; il consiste à ajouter un bit supplémentaire à un certain nombre de bits de données appelé *mot de code*. Souvent, il y a 7 bits réservés au message auquel on ajoute 1 bit de parité **égal à la somme des 7 bits modulo 2**. Cela signifie que si le nombre de bits égaux à 1 du message est impair, alors on ajoute 1 comme bit de contrôle, sinon on ajoute 0. L'ensemble ainsi constitué est donc codé sur un octet (8 bits). L'exemple le plus classique est le codage des caractères ASCII qui sont au nombre de 128 et sont donc codés sur 7 bits, auquel on peut ajouter le bit de parité, pour finalement former un octet.

Dans la suite, on n'aura pas à vérifier que les listes passées en entrée des fonctions sont constituées de bits, et donc de 0 ou de 1.

❶ Ecrire le script Python d'une fonction `somme_modulo(L)` prenant en entrée une liste de bits L non vide (inutile de le vérifier), non nécessairement de longueur 7, et renvoyant la somme de ses bits, modulo 2.

❷ En déduire une fonction `rajoute_bit_parite(L)` prenant en entrée une liste L **non vide**, et renvoyant cette même liste avec un bit supplémentaire à la fin correspondant au bit de parité. **Attention :** la liste L ne devra pas être modifiée.

❸ On suppose qu'une liste (non nécessairement de 8 bits) constituée de 0 et de 1 codée par bit de parité a subi au plus une erreur lors de la transmission. Montrer que pour savoir si elle a subi une erreur ou non, il suffit de calculer la somme de ses bits modulo 2. Si le résultat est 0, il n'y a pas eu d'erreur, sinon il y en a eu une.

❹ En déduire une fonction `decodage_parite` prenant en entrée une liste de bits (de taille au moins 2, inutile de le vérifier) correspondant à la liste reçue après transmission d'une liste codée par bit de parité ayant subi au plus une erreur, et qui :

- s'il y a eu une erreur de transmission, affiche un message d'erreur à l'écran.
- sinon, retourne **uniquement** une **nouvelle liste** dont les éléments sont ceux de L, le bit de parité supprimé.

❺ Le codage par bit de parité paire permet-il de corriger une erreur ? Permet-il par ailleurs de détecter deux erreurs ?

II Codage par répétition

Le code de répétition est une solution simple pour se prémunir des erreurs de communication dues au "bruit" (perturbation liée à l'environnement ou la structure même du système de communication). C'est une technique de codage de canal, c'est-à-dire un code correcteur.

Coder une liste par répétition consiste à répéter un certain nombre de fois k chaque bit de la liste. Par exemple, avec la liste L suivante, et $k = 3$, on obtient :

$$L = [1, 1, 0, 0, 1] \quad \longrightarrow \quad [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1]$$

❶ Ecrire une fonction `repetition(L,k)` prenant en entrée une liste de bits L, non vide, et un entier k, et retournant la liste correspondant à la répétition k fois de chaque bit de L. **Attention :** on veillera là-encore à conserver intacte la liste originelle L.

Le décodage après transmission se fait en considérant les bits de L par paquets de k bits (k étant connu du destinataire). On regarde simplement quel bit apparaît le plus dans chaque paquet. La figure ci-dessous montre le fonctionnement sur la liste prise en exemple : les bits entourés correspondent à deux erreurs de transmission mais le message décodé est correct. Par exemple avec la liste L précédente et $k = 3$

$$\underbrace{1\,1\,\boxed{0}}_{1}\quad\underbrace{1\,1\,1}_{1}\quad\underbrace{0\,0\,0}_{0}\quad\underbrace{0\,0\,0}_{0}\quad\underbrace{\boxed{0}\,1\,1}_{1}$$

Le décodage a une petite différence suivant la parité de k :

- si k est impair, il y a forcément un élément majoritaire (0 ou 1) sur chaque paquet de k bits.
- si k est pair, il se peut qu'il y ait autant de bits à 0 que de bits à 1 sur chaque paquet de k bits. Dans ce cas, le décodage est rendu impossible !

❷ Ecrire une fonction `decodage_majoritaire(L,k)` prenant en entrée une liste L de *bits*, dont la longueur qu'on notera n ici est supposée être un multiple de k, ce que le code devra vérifier (et renvoyer un message d'erreur si nécessaire), et renvoyant une liste de taille $\frac{n}{k}$, correspondant au décodage majoritaire de chaque paquet de k bits de L. Si on trouve un groupe de k bits contenant autant de 0 que de 1, on affichera une erreur à l'écran, et on ne renverra aucune liste.

❸ Justifier que le code de répétition (avec facteur de répétition k) permet de détecter au moins $\left\lfloor \frac{k}{2} \right\rfloor$ erreurs et d'en corriger au moins $\left\lfloor \frac{k-1}{2} \right\rfloor$, **ceci dans chaque bloc**.

III Le contrôle de parité croisé (ou LRC pour *Longitudinal Redundancy Checking*)

III.1 Principe

Supposons que la liste L que l'on souhaite coder soit de taille n^2 avec n un entier. On peut alors répartir ces éléments en n listes elles-mêmes de taille n, et voir la liste L comme un tableau en deux dimensions. La figure suivante présente cette visualisation avec une liste L contenant 9 éléments, répartis en 3 listes de taille 3 :

$$L = [[0,1,0],[1,1,0],[0,1,1]] \quad \text{qu'on peut visualiser ainsi :} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Pour i et j deux entiers entre 0 et $n-1$, $L[i]$ est la $i^{\text{ème}}$ liste de L (correspondant à la i$^{\text{ème}}$ ligne du tableau), et donc l'élément $L[i][j]$ est le bit sur la $i^{\text{ème}}$ ligne et la $j^{\text{ème}}$ colonne (lignes et colonnes sont numérotées à partir de 0 comme toujours en Python).
Le code de parité croisé consiste à rajouter un bit à chaque ligne et à chaque colonne, correspondant au bit de parité de chaque ligne/colonne. On ajoute également un bit de parité en bas à droite (correspondant à la fois à la parité de la colonne et de la ligne auquel il appartient). On obtient ainsi le tableau suivant :

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

❶ Montrer mathématiquement que le bit de parité en bas à droite (coordonnées $[n,n]$ dans un tableau en Python) fonctionne aussi bien pour sa colonne que pour sa ligne.

❷ On considère le tableau suivant, correspondant au tableau obtenu après transmission d'un tableau de taille 4×4 auquel on a rajouté les bits de parité sur les lignes et les colonnes comme décrit précédemment, pour obtenir un tableau de taille 5×5 :

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

On assure qu'il y a eu au plus une erreur de transmission. Si il y en a eu une, donnez l'emplacement du bit corrompu, et sinon, justifiez qu'il n'y a pas eu d'erreur.

❸ Justifiez que le contrôle de parité croisé permet de détecter et de corriger une erreur. Peut-il à coup sûr détecter 2 erreurs ? En corriger 2 ? On justifiera soigneusement les réponses.

III.2 Mise en oeuvre

❶ **Codage par contrôle de parité croisé**

Ecrire une fonction `rajoute_parite_croise(T)` prenant en entrée un tel tableau (sous forme de listes de listes). En notant $n\times n$ ses dimensions (n est classiquement accessible par `len(L)`), la fonction renvoie un tableau (sous forme de listes de listes) de taille $(n+1)\times(n+1)$ correspondant à l'ajout du contrôle de parité croisé.

❷ **Décodage par contrôle de parité croisée**

a· Ecrire une fonction `parite_lignes(T)` prenant en entrée une liste de listes T correspondant à un tableau codé par contrôle de parité croisé, et renvoyant la liste des indices des lignes pour lesquelles la parité n'est pas respectée (on renverra une liste vide si la parité est respectée sur toutes les lignes).

b· Ecrire de même une fonction pour les colonnes `parite_colonnes(T)`.

c· Ecrire une fonction `decodage_parite_croise(T)`, prenant en entrée une liste de listes T supposée être encodée par contrôle de parité croisé et récupérée après transmission (donc ayant subi des erreurs éventuelles) et :

- retournant la liste initiale après décodage, si le décodage paraît possible (on suppose que le nombre d'erreurs ayant eu lieu est le plus petit possible pouvant produire la liste T après correction)
- affichant un message d'erreur à l'écran indiquant si on sait qu'au moins un certain nombre d'erreurs a eu lieu mais qu'on est incapable de décoder.