Devoir Surveillé 3 : Correction

1 Un problème de rendu de monnaie

On s'intéresse ici au nombre de manière différentes de rendre une somme n en fonction d'une liste de type de pièces disponible stockées dans une liste L. On considérera que l'on dispose de suffisamment de pièces de chaque type pour rendre la somme n.

On va écrire une fonction `pieces(n,L)` qui va renvoyer le nombre de manières différentes d'obtenir n avec des pièces ayant des valeurs dans L.

Question 1. Montrer que `pieces(3,[1,2])` vaut 2
Que vaut `pieces(2,[0.5,1])` ?

On peut écrire $3 = 3 \times 1$ ou $3 = 2 + 1$, ce qui donne deux possibilités. Dans ce problème, il est clair que l'ordre est indifférent. On peut donc noter les pièces par ordre de valeur décroissante.

On peut écrire $2 = 2 \times 1$ ou $2 = 1 + 2 \times 0.5$ ou encore $2 = 4 \times 0.5$. On a donc `pieces(2,[0.5,1])`=3.

Il est par contre clair que si $0 \in$ `L`, le nombre de possibilités devient infini ! En effet un appel à `pieces(2,[2,0])` pourrait renvoyer toute écriture de la forme $2 = 1 \times 2 + m \times 0$ avec $m \in \mathbb{N}$

Question 2. Compléter le programme suivant :

```
def pieces(n,L):
      if n==0: # (1)
            return 1
      elif L == [] or n < 0:
            return 0 # (2)
      else:
            return pieces(n, L[1:]) + pieces(n-L[0], L) # (3)
```

(1) Quand on arrive à $n = 0$, cela correspond à un succès dans le rendu de monnaie : on a épuisé n en puisant dans la liste L.
(2) Quand on arrive à une situation d'impasse, ce qui est le cas si l'on doit rendre la monnaie restante sans plus disposer d'aucune pièce (liste vide) ou que les pièces disponibles sont de valeurs supérieures à la somme à rendre, on renvoie 0, c'est-à-dire que l'on n'ajoute pas 1 à l'addition récursive.
(3) On partitionne les rendus de monnaie en deux parties : les situations où l'on n'utilise pas la valeur `L[0]` et celles où l'on prend une pièce de valeur `L[0]`.

Question 3. Montrer que ce programme se termine.

Le programme se termine... si $0 \notin L$. C'est la mise en garde de la première question. On note qu'à chaque appel récursif, la quantité `n + len(L)` est strictement décroissante et que les deux conditions de sorties permettent de ne pas créer d'appel infni : quoi qu'il arrive, on enlève à L des éléments et `len(L)` est une suite décroissante positive. Les valeurs successives de `n` sont aussi décroissantes (au sens large) et il peut arriver que la soustraction `n-L[0]` fasse passer la valeur en-dessous de 0, mais le cas est traité.

2 Transformations de listes récursives

Question 4. Écrire une fonction récursive `swap2` qui prend en argument une liste `L` et qui renvoie cette liste où l'ordre des éléments est modifié de sorte à ce que cette liste contienne : le second élément de L puis le premier, puis le quatrième, puis le troisième, puis le sixième, puis le cinquième etc...

Ainsi `swap2([1,4,9,16,25,36,49])` renverra `[4,1,16,9,36,25,49]`

```
def swap2(L):
      if len(L) < 2:
            return L
      return [L[1], L[0]] + swap2(L[2:])
```

Question 5. Réécrire cette fonction dans un cas plus général : écrire une fonction récursive `swap` qui prend en arguments une liste `L` et un entier strictement positif `k` qui renvoie la liste constituée du $k^{\text{ème}}$ élément de L, puis du $k-1^{\text{ème}}$, ..., puis du premier, puis du $2k^{\text{ème}}$ ... du $k+1^{\text{ème}}$ etc...

Ainsi `swap([1,4,9,16,25,36,49],3)` renverra `[9,4,1,36,25,16,49]`

```
def swapp(L, k):
if len(L) < k:
return L
M = L[:k]
M.reverse()
return M + swapp(L[k:], k)
```

```
def swap(L, k):
if len(L) < k:
return L
return L[k-1::-1] + swap(L[k:], k)
```

L'énoncé est un peu ambigu en ce qu'il ne précise pas s'il faut inverser la queue de la liste (les `len(L)%k` éléments qui restent à la fin après avoir épuisé les découpages possibles en sous-listes de longueur k. On a choisi ici de la laisser à l'identique. S'il faut l'inverser aussi, il suffit de remplacer le `return L` de la condition de sortie par `return L[::-1]`.

La première solution utilise la méthode `reverse`, qui retourne une liste en place. Elle nécessite donc de commencer par extraire la sous-liste à inverser, puis de l'inverser, avant de pouvoir enfin écrire un `return`. En effet, comme c'est une méthode, cela n'aurait aucun sens d'écrire `return L.reverse()`.

La deuxième solution utilise toute la puissance du slicing et de ses trois arguments. On aurait pu aussi remplacer `L[k-1::-1]` par `list(reversed(L[:k]))`, mais cette solution, quoique rapide à l'écriture, est lente à l'exécution car elle commencer par créer un itérateur à partir duquel elle crée une liste.

Question 6. On considère dans cette question que l'affectation ou l'ajout d'un élément à une liste a un coût de 1, et qu'un slicing de liste a un coût égal à la longueur de la liste slicée : ainsi `L[a:b]` a un coût de $b-a$ et `L[:]` a un coût de n=`len(L)`

Exprimer et justifier la complexité (asymptotique) de l'algorithme écrit précédemment en fonction de n et de k. Si besoin, on pourra considérer que n est grand devant k.

Prenons la variante avec le slicing. Chaque passage dans la fonction récursive a un coût de $k+2$, avec k pour le slicing, 1 pour la comparaison et 1 pour l'addition. Il y a $\lfloor n/k \rfloor$ passages dans la boucle amenant la transformation d'une liste de longueur k, le dernier passage (qui a lieu si n n'est pas multiple de k) a un coût de 1 (pas d'inverse) ou de $n\%k+1$ dans le cas contraire. Bref, on peut dire que le coût est en $\mathcal{O}(k \times (n/k)) = \mathcal{O}(n)$ et ne dépend donc pas de k.

3 Algèbre linéaire et numpy

Cet exercice sera traité en utilisant le type `array` du module `numpy`

Question 7. Écrire une fonction `transpose` qui prend en argument une matrice M et renvoie la matrice transposée M^T.

```
def transpose(M): # boucle
n, p = np.shape(M)
N = np.zeros((p,n))
for i in range(p):
for j in range(n):
N[i,j] = M[j,i]
return M
```

```
def transpose(M): # compréhension
n, p = M.shape
return np.array([[M[i,j] for j in range(p)] for i in range(n)])
```

Question 8. Écrire une fonction `test` qui prend en argument un couple de matrices (A, M), renvoie un message d'erreur si A et M ne sont pas des matrices carrées de même taille et la matrice $AM + MA^T$ sinon.

```
def test(A, M):
      chaipe = A.shape
      assert M.shape == chaipe and chaipe[0] == chaipe[1]
      return np.dot(A, M) + np.dot(M, transpose(A))
```

Question 9. Écrire une fonction `base_canonique` d'argument un entier naturel n et qui renvoie une liste dont les éléments sont, dans l'ordre usuel, les éléments de la base canonique de $\mathcal{M}_n(\mathbb{R})$.

```
def base_canonique(n):
      L = []
      for i in range(n):
            for j in range(n):
                  M = np.zeros((n,n))
                  M[i,j] = 1
                  L.append(M)
      return L
```

Question 10. Écrire une suite d'instructions définissant la matrice $A = \begin{pmatrix} 1 & 2 & -3 \\ 4 & 2 & 0 \\ 5 & 6 & 1 \end{pmatrix}$ et générant la matrice de l'application $M \longmapsto AM + MA^T$ dans la base canonique de $\mathcal{M}_n(\mathbb{R})$.

`A = np.array([1, 2, 3, 4, 2, 0, 5, 6, 1]).reshape(3,3)` ou bien
`A = np.array([[1, 2, -3], [4, 2, 0], [5, 5, 1]])`

```
bc = base_canonique(3)
col = [test(A,M) for M in bc]
Mat = transpose(np.array([c.reshape(9) for c in col]))
```

4 Traitement d'images

Une image en couleur acquise par une caméra est constituée de trois couches (RGB).

Les données de l'image sont stockées dans un tableau à trois dimensions : la première dimension correspond à la couleur (rouge, vert ou bleu, indice 0, 1 ou 2), la seconde correspond à la coordonnée selon x et la troisième à la coordonnée selon y. Ainsi, les dimensions du tableau sont : $3 \times m \times n$ où $m = 800$ et $n = 600$.

La valeur associée à chaque pixel est un entier compris entre 0 et 255.

Question 11. Donner la quantité de mémoire nécessaire en octets pour stocker le tableau représentant l'image émise par la caméra en justifiant le codage retenu pour un pixel d'une couche.

Chaque pixel est codé par un triplet d'entiers non signés de l'intervalle $[\![0, 255]\!] = [\![0, 2^8 - 1]\!]$. On choisit donc le format `uint8`, qui permet de coder chacune des trois couleurs sur un octet.

Pour une image, il faut donc $800 \times 600 \times 3 = 1440000 = 1,44$ Mo.

Question 12. Dans le cadre du traitement des images acquises, la première étape est de convertir l'image en couleur en niveaux de gris. La méthode consiste à rechercher le maximum et le minimum pour chaque pixel sur les trois couches, puis à faire la moyenne de ces maxima et minima et ainsi obtenir la valeur du nouveau pixel en niveaux de gris.

On note `imagecolor` le tableau représentant une image en couleur.

Écrire une fonction `grayscale(imagecolor)` qui renvoie une image en niveaux de gris (qui sera notée image dans l'algorithme principal), sous forme de tableau à deux dimensions de taille m × n contenant des valeurs entières comprises entre 0 et 255, en suivant l'algorithme décrit ci-dessus. Il est possible d'utiliser les fonctions `min` et `max` de Python.

```
def grayscale(imagecolor):
      image = np.zeros((800, 600)).astype('uint8')
      for i in range(800):
            for j in range(600):
                  couleur = imagecolor[0, i, j], imagecolor[1, i, j], imagecolor[2, i, j]
                  m, M = min(couleur), max(couleur)
                  image[i, j] = (m+M)/2
      return image
```

Il faut bien déclarer le tableau comme de type `uint8` et non comme un `np.zeros((800, 600))`, qui sera automatiquement de type `float64` car, en ce cas, l'affectation `image[i, j] = np.uint8((m+M)/2)` ne sera pas respectée. Un tableau numpy ayant nécessairement tous ses éléments de même type, Python respectera le type flottant initial, les affectations étant individuelles.

5 Bijections de $\mathbb{N}$ vers $\mathbb{N}^2$

On construit une bijection de $\mathbb{N}$ sur $\mathbb{N}^2$ en numérotant les points entiers du quart de plan supérieur droit diagonale par diagonale et de bas en haut : ainsi, les premiers éléments sont $u_0 = (0, 0)$, $u_1 = (1, 0)$, $u_2 = (0, 1)$, $u_3 = (2, 0)$, $u_4 = (1, 1)$, $u_5 = (0, 2)$, $u_6 = (3, 0)$, etc...

Question 13. Écrire une fonction récursive prenant en argument (a, b) et calculant l'indice n tel que $u_n = (a, b)$.

```
def n2n(a,b):
      if a + b == 0:
            return 0
      elif b == 0:
            return n2n(0, a-1) + 1
      else:
            return n2n(a+1, b-1) + 1
```

Question 14. Coder également la fonction réciproque, c'est-à-dire la fonction qui à n associe u_n. On commencera par montrer que, pour $T_p = \dfrac{p(p+1)}{2}$, si $u_n = (x_n, y_n)$ et $T_p \leqslant n < T_{p+1}$, alors $x_n + y_n = T_p$.

```
from math import sqrt
def nn2(n):
      p = int((-1+sqrt(8*n+1))/2)
      Tp = p*(p+1)//2
      return (p -(n-Tp), n-Tp)
```