

# DS4 Informatique 802-803 Correction

## EXERCICE 1: Problème du sac à dos

Un voleur entre par effraction dans une cabane de jardin. Il y trouve  $n$  objets de valeur mais son sac à dos ne peut emporter qu'un poids maximal  $W$ .

Il souhaite choisir les objets qu'il va mettre dans son sac à dos de telle sorte que la somme de leur valeur soit maximale et que le poids total ne dépasse pas  $W$ .

Pour cela il utilise l'algorithme glouton suivant: on ajoute en priorité les objets « les plus rentables », « plus rentable » ne signifiant pas « plus grande valeur » mais « plus grande valeur comparativement à la masse ».

Pour chaque objet  $i$ , notons  $w_i$  son poids et  $v_i$  sa valeur. L'algorithme est alors le suivant:

Pour chaque objet  $i$  calculer sa rentabilité  $\frac{v_i}{w_i}$

Trier les objets par ordre de rentabilité décroissante

En respectant cet ordre placer les objets un par un dans le sac à dos: un objet est placé à condition qu'il ne fasse pas dépasser le poids maximal  $W$ , sinon on passe à l'objet suivant dans la liste triée.

**Q1.** Dans la cabane de jardin le voleur trouve les objets suivants:

Objet	1	2	3	4	5
Valeur	4	10	14	4	9
Poids	2	4	20	15	7

Avec ces objets appliquer à la main l'algorithme glouton avec un sac à dos pouvant emporter un poids maximal  $W = 30$ .

**On ne demande pas d'écrire un programme Python.**

La liste des rentabilités est  $[2, 2.5, 0.7, 4/15, 9/7]$  donc la liste des objets par rentabilité décroissante est  $[2, 1, 5, 3, 4]$ .

On place donc l'objet 2 dans le sac à dos ce qui fait un poids de 4 et une valeur de 10. Ensuite on place l'objet 1 ce qui fait un poids total de 6 et une valeur totale de 14. Puis on place l'objet 5 ce qui fait un poids total de 13 et une valeur totale de 23. Il n'y a pas de place pour l'objet 3. Par contre on peut prendre l'objet 4 pour arriver à un poids final de 28 et une valeur finale de 27.

**Q2.** La solution donnée par l'algorithme glouton dans l'exemple précédent est-elle optimale?

Non car on peut prendre les 3 premiers objets dont le poids total est de 26 et on a alors une valeur emportée égale à 28. C'est mieux que la solution donnée par l'algorithme glouton!

## EXERCICE 2: Traitement d'images

On suppose que les variables `photo1` et `photo2` sont des tableaux Numpy contenant une image de taille  $240 \times 480$  au format RGB (chaque couleur est codée par un entier entre 0 et 255).

**Q1.** Pour augmenter la luminosité d'une image, il suffit d'ajouter (ou soustraire) à toutes les valeurs un même nombre sans dépasser la valeur maximale qui est de 255 si on ajoute (ou minimale qui est 0 si on soustrait).

Modifier le script suivant pour augmenter la luminosité de 80 dans l'image `photo1` (l'utilisation des fonctions `min()` et `max()` est autorisée):

```
for ligne in range(240):
    for colonne in range(480):
        for couleur in range(3):
            photo1[ligne,colonne,couleur] = min( 255, photo1[ligne,colonne,couleur] + 80 )
```

**Q2.** On peut, à partir de deux images, créer un mélange des deux. Pour cela, on commence par choisir dans quelle proportion on veut les mélanger (par exemple 60% de la première et donc 40% de la seconde). Ensuite, il suffit de prendre, pour chaque couleur de chaque pixel, 60% de la valeur de la première image et 40% de la valeur de la seconde image.

Ainsi, si par exemple le premier pixel de la première image est de couleur (10, 20, 30) et le premier pixel de la seconde image est de couleur (100, 100, 100) alors l'image mélangée sera de couleur  $(0.6 \times 10 + 0.4 \times 100, 0.6 \times 20 + 0.4 \times 100, 0.6 \times 30 + 0.4 \times 100)$  c'est à dire (46, 52, 58).

Ecrire un script qui mélange `photo1` et `photo2` avec une proportion de 60% de la première et 40% de la seconde, et enregistre le résultat dans une variable `photo3` de type tableau Numpy.

On rappelle qu'on peut créer un tableau Numpy de taille  $240 \times 480 \times 3$  rempli de 0 stocké dans une variable `t` avec l'instruction `t = zeros((240, 480, 3))`.

```
photo3 = zeros((240, 480, 3))
for ligne in range(240):
    for colonne in range(480):
        for couleur in range(3):
            photo3[ligne,colonne,couleur] = 0.6*photo1[ligne,colonne,couleur] + 0.
            ↪ 4*photo2[ligne,colonne,couleur]
```

## EXERCICE 3: Récursivité

**Q1.** Ecrire une fonction **récursive** `factorielle(n)` qui prend en entrée un entier `n` et renvoie en sortie la valeur de `n!`.

```
def factorielle(n):
    if n == 0:
        return 1 # close d'arrêt
    else:
        return n*factorielle(n-1) # appel récursif
```

**Q2.** Ecrire une fonction **itérative** `factorielle2(n)` qui prend en entrée un entier `n` et renvoie en sortie la valeur de `n!`.

```
def factorielle2(n):
    x = 1
    for k in range(1, n+1): # k va de 1 à n
        x = x*k
    return x
```

**Q3.** Ecrire une fonction **récursive** `miroir(L)` qui prend en entrée une liste `L` et renvoie en sortie une liste composée des éléments de `L` mais dans l'ordre inverse.

Par exemple `miroir([1, 2, 3, 4])` renvoie `[4, 3, 2, 1]`.

```
def miroir(L):
    if len(L) == 0:
        return []
    else:
        return miroir(L[1:]) + [ L[0] ]
```

**Q4.** Si `x` est un réel et `n` est un entier alors: 
$$x^n = \begin{cases} (x^{n/2})^2 & \text{si } n \text{ est pair} \\ x \times (x^{(n-1)/2})^2 & \text{si } n \text{ est impair} \end{cases}$$

En déduire une fonction **récursive** `puissance_rapide(x,n)` qui prend en entrée un réel `x` et un entier naturel `n` et renvoie la valeur de `xn` calculée avec la formule précédente.

```
def puissance_rapide(x,n):
    if n == 0:
        return 1 # close d'arrêt
    elif n%2 == 0:
        return ( puissance_rapide(x, n//2) )**2 # appel récursif
    else:
        return x *( puissance_rapide(x, (n-1)//2) )**2 # appel récursif
```

**Q5.** Ecrire une fonction **récursive** `permutation(L)` qui prend en entrée une liste `L` et renvoie en sortie la liste de toutes ses permutations. Par exemple `permutation([1, 2, 3])` renvoie la liste de listes `[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]`

```
def permutation(L):
    if len(L) == 0: # close d'arrêt:
        return [[]] # attention c'est une liste de listes (et non pas une liste)
    else:
        sortie = []
        for k in range(len(L)): # on va donner les permutations qui commencent par L[k]
            for liste in permutation(L[:k]+L[k+1:]): # liste est une permutation de L privée de L[k]
                liste2 = [ L[k] ] + liste # on ajoute L[k] en tête de liste
                sortie.append( liste2 ) # liste2 est ajoutée dans la liste des permutations
        return sortie
```