Correction du TP n°2: Le problème du Sac à Dos

4 juillet 2019

I Retour sur les Tris

1. Ecrire une fonction **extraire** qui accepte en paramètres, une liste de liste **l** et un entier **n** dans [0 ; 2] .La fonction doit retourner la liste triée en fonction du n-ième élément de chaque sous-liste. On pourra choisir le tri insertion par exemple.

Console Python

```
>>> L= [ [1, 5, 6], [8, 10, 2], [3, 3, 5],[4, 8, 1] ]
>>> extraire(L,0)
[[1, 5, 6], [3, 3, 5], [4, 8, 1], [8, 10, 2]]
>>> extraire(L,1)
[[3, 3, 5], [1, 5, 6], [4, 8, 1], [8, 10, 2]]
>>> extraire(L,2)
[[4, 8, 1], [8, 10, 2], [3, 3, 5], [1, 5, 6]]
```

```
1 def extraire(t,n):
2     for i in range(1,len(t)):
3         v = t[i][n]
4         j = i-1
5         while(j>=0 and t[j][n]>v):
6             t[j+1],t[j] = t[j],t[j+1]
7             j = j-1
8         t[j+1][n] = v
9     return t
```

2. Pourquoi le code précédent ne fonctionne-t-il pas (tel quel) avec une liste de tuples ?

Solution: Le tuple est un objet que l'on ne peut pas modifier donc une instruction comme celle de la ligne 8 par exemple n'est pas autorisée. Il faudrait par exemple, copier la liste de tuples en liste de liste puis trier et retourner une liste de tuples.

3. A l'aide de l'exercice 7, expliquer le sens de l'instruction `List.sort(key = lambda a : a[1])`

Solution: "list.sort() and sorted() have a key parameter to specify a function to be called on each list element prior to making comparisons." [...]accept a reverse parameter with a boolean value. This is used to flag descending sorts.

Dès lors, on peut appeler une fonction lambda(nom donné à une fonction nommée "à la volée") pour trier des objets à un certain indice dans l'élément courant (si cet élément est une liste, un tuple,...) En conséquence, le tri sur a[1] signifie que l'on s'intéresse au deuxième élément du tuple. Par ailleurs, reverse=True indique qu'il s'agit d'un tri "décroissant".

II Algorithme du sac à dos

A Critère de Poids

Nous allons écrire un premier algorithme glouton de critère **"placer d'abord dans le sac, les objets les plus lourds."** En cas d'égalité, on prendra "au hasard" un objet parmi les indécis. On pourra utiliser la méthode **list.sort()** comme définie sur la documentation officielle Python

4. Ecrire une fonction **gloutonP** qui accepte deux paramètres **l** et **maxi**, respectivement une liste de tuples (valeur en €, poids en kg) et un entier correspondant au poids maximal supporté par le sac. La fonction doit donc retourner un triplet **reponse, valeur, poids** comme dans l'exemple ci-dessous :

Console Python

```
>>> liste=[(7,13), (4,12), (3,8), (3,10)]
>>> print(gloutonP(liste,30))
([1, 1, 0, 0], 11, 25)
```

Solution: (voir Annexe B) On crée une copie de la liste de tuples **lig(2 et 3)** puis on trie la liste "temporaire" selon le 2e élément (tup[1]) dans l'ordre décroissant (reverse=True) ; **lig(8)** on rentre dans une boucle "de la longueur" du tableau. Si le poids actuel du sac plus le poids du prochain objet est inférieur ou égal au poids maxi **lig(9)**, alors on ajoute cet objet à la solution **lig(10)**, puis on augmente le poids et la valeur du sac **lig(11 et 12)**. Enfin, on réalise une compréhension en utilisant la liste initiale pour afficher soit 1 soit 0 selon que l'élément est dans la solution ou pas.

Le fichier **Datas.txt** contient un ensemble de données correspondant à 5 magasins fictifs : t4, t7, t8, t10 et t15. Chaque ligne du fichier contient, le nom du magasin, une liste d'objets et le poids maximal autorisé dans le sac du voleur.

5. A l'aide de la fonction **perf_counter** du module **time**, calculer la durée d'exécution sur les ensembles de données fournis dans le fichier Datas.txt. Les durées seront obtenues en réalisant une moyenne sur 100 appels de la fonction **gloutonP**. Noter vos résultats dans les tableaux en annexe A (avec 4 chiffres significatifs sur le temps).

Solution: (voir Annexe A)

B Critère de Valeur

Pour notre deuxième algorithme glouton, le critère retenu est **"placer d'abord dans le sac, les objets de plus grande valeur"**. Ecrire une fonction **gloutonV** qui accepte les mêmes paramètres que gloutonP et retourne le triplet **reponse, valeur, poids** comme dans l'exemple ci-après ; réaliser ensuite les mêmes tests à l'aide du fichier **datas.txt** pour compléter les tableaux de l'annexe A.

Console Python

```
>>> liste=[(7,13), (4,12), (3,8), (3,10)]
>>> gloutonV(liste,30)
([1, 1, 0, 0], 11, 25)
```

C Critère de Valeur Pondérée par le Poids

Pour notre dernier algorithme glouton, le critère retenu est **"placer d'abord dans le sac, les objets dont le rapport $\frac{valeur}{poids}$ est le plus élevé**. Ecrire une fonction **gloutonVPP** qui accepte les mêmes paramètres que gloutonP et retourne le triplet **reponse, valeur, poids** comme dans l'exemple ci-après ; réaliser ensuite les mêmes tests à l'aide du fichier **datas.txt** pour compléter les tableaux en annexe A.

Solution: 6. - 7. (voir Annexe B) Algorithmes identiques à gloutonP. Seule la clé du tri change : la liste "temporaire" est triée soit selon le 1er élément (tup[0]) soit selon le rapport $\frac{tup[0]}{tup[1]}$ et dans l'ordre décroissant (reverse=True).

Console Python

```
>>> liste=[(7,13), (4,12), (3,8), (3,10)]
>>> print(gloutonVPP(liste,30))
([1, 1, 0, 0], 11, 25)
```

D Comparaison des trois critères

6. Comparer les durées d'exécution des trois algorithmes gloutons. Commenter.

Solution: (voir Annexe A) gloutonP, gloutonV et gloutonVPP possèdent des durées d'exécution comparables (de l'ordre de $1 \times 10^{-5} s$) et ceci quel que soit n (pour notre petit échantillon). Les algorithmes gloutons semblent posséder une certaine "stabilité en temps".

7. Discuter du choix du critère de sélection des algorithmes gloutons en fonction de n.

Solution: (voir Annexe A) gloutonP semble le plus mauvais des trois car il ne fait jamais mieux que les deux autres ; gloutonV semble le meilleur pour des petits échantillons mais à partir de n=10 gloutonVPP domine les deux autres. Il faudrait toutefois pouvoir réaliser des tests sur des valeurs de n bien supérieures pour "solidifier" notre hypothèse.

A Performances des différentes algorithmes

Durée d'exécution (en s)* de différents algorithme en fonction du nombre d'objets pour le KP					
Algorithme	n=4	n=7	n=8	n=10	n=15
GLOUTONP	8.490e-6	1.284e-5	1.967e-5	1.857e-5	2.559e-5
GLOUTONV	1.059e-5	8.202e-6	1.487e-5	1.209e-5	3.243e-5
GLOUTONVPP	2.071e-5	7.543e-6	1.990e-5	1.431e-5	2.716e-5
Algorithme Naif	1.291e-4	2.129e-3	4.348e-3	2.100e-2	1.137

TABLE 1 – Durée d'exécution

* Calculs effectués sur 1000 itérations pour chaque fonction ; MacBook Pro IntelCore i5 2.4 GHz double coeur

Réponses** des différents algorithmes en fonction du nombre d'objets pour le KP					
Algorithme	n=4	n=7	n=8	n=10	n=15
GLOUTONP	([1,1,0,0], 11,25)	([0, 1, 0, 1, 0, 0, 1], 1735, 169)	([1, 0, 0, 0, 1, 0, 1, 0], 65, 102)	([0, 0, 0, 0, 0, 0, 1, 0, 1, 0], 154, 152)	([1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1], 1315, 682)
GLOUTONV	([1,1,0,0], 11, 25)	([0, 1, 0, 1, 0, 0, 1], 1735, 169)	([1, 1, 1, 1, 0, 1, 0, 0], 280, 102)	([1, 0, 0, 1, 0, 0, 0, 0, 1, 0], 247, 156)	([1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1], 1315, 682)
GLOUTONVPP	([1, 0, 1, 0], 10,21)	([1, 1, 1, 0, 0, 0, 0], 1478, 140)	([1, 1, 1, 1, 0, 0, 0, 1], 266, 73)	([1, 1, 1, 1, 0, 1, 0, 0, 0, 0], 309, 165)	([0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0], 1441, 740)
Sol. Optimale	([1,1,0,0], 11, 25)	([0, 1, 0, 1, 0, 0, 1], 1735, 169)	([1, 1, 1, 1, 0, 1, 0, 0], 280, 102)	([1, 1, 1, 1, 0, 1, 0, 0, 0, 0], 309, 165)	([0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0] , 1458, 749)

TABLE 2 – Comparaison des réponses obtenues

** Code couleur pour les algorithmes gloutons
Réponse optimale ou réponse approchée la plus proche
Réponse approchée "intermédiaire"
Réponse approchée "éloignée"

B Question 4,6,7 - Proposition d'algorithmes

```python
def gloutonP(l,maxi):
    tmp=[x for x in l]
    tmp.sort(key=lambda tup: tup[1],reverse=True)
    valeur=0
    poids=0
    i=0
    solution=[]
    while i<len(tmp):
        if (poids+tmp[i][1]) <=maxi:
            solution.append(tmp[i])
            poids +=tmp[i][1]
            valeur+=tmp[i][0]
        i+=1
    reponse=[1 if x in solution else 0 for x in l]
    return reponse,valeur,poids
```

```python
def gloutonV(l,maxi=30):
    tmp=[x for x in l]
    l.sort(key=lambda tup: tup[0],reverse=True)
    valeur=0
    poids=0
    solution=[]
    i=0
    while i<len(tmp):
        if (poids+tmp[i][1]) <=maxi:
            solution.append(tmp[i])
            poids +=tmp[i][1]
            valeur+=tmp[i][0]
        i+=1
    reponse=[1 if x in solution else 0 for x in tmp]
    return reponse,valeur,poids
```

```python
def gloutonVPP(l,maxi=30):
    tmp=[x for x in l]
    l.sort(key=lambda tup: tup[0]/tup[1],reverse=True)
    valeur=0
    poids=0
    solution=[]
    i=0
    while i<len(tmp):
        if (poids+tmp[i][1]) <=maxi:
            solution.append(tmp[i])
            poids +=tmp[i][1]
            valeur+=tmp[i][0]
        i+=1
    reponse=[1 if x in solution else 0 for x in tmp]
    return reponse,valeur,poids
```