## Table des matières

# Le problème du sac à dos

## 1 Introduction

Un voleur pénètre dans la maison d'un riche trader. Les objets les plus intéressants sont une collier de perles, un candelabre en argent, le livre « algorithmes » de Cormen, Leiserson, Rivest, et Stein dédicacé par les auteurs, et une fourrure de glomorphe à rayures. Comme il doit rester léger pour escalader la façade de l'immeuble, il n'a pris qu'un petit sac à dos pouvant contenir au maximum 8kg. Le tableau suivant donne le poids et la valeur (en centaines d'euros) des différents objets :

| objet | poids (kg) | valeur (100€) |
|---|---|---|
| Collier | 1 | 15 |
| Candelabre | 5 | 10 |
| Cormen | 3 | 9 |
| Glomorphe | 4 | 5 |

Le voleur s'interroge sur les objets qu'il doit mettre dans son sac à dos. On envisage trois stratégies :

1. *Le voleur est « glouton » :* L'algorithme glouton consista à prendre en priorité les objets de plus grande valeur. Quels objets prendra-t-il s'il suit cette méthode ? Le choix est-il optimal ?
   *solution :* Il prend le collier et le candelabre : ensuite il n'a plus de place. Cela lui rapporte 2500€. S'il avait pris le collier, le Cormen et la peau de glomorphe, il aurait gagné 2900€. Sa stratégie n'est donc pas optimale.
2. *Le voleur a le temps :* Supposons que le voleur décide de caculer toutes les combinaisons d'objet possibles. Il lui faut 5s pour analyser chaque possibilité (voir si elle est possible et calculer combien elle lui rapporte). Combien de temps lui faudra-t-il ?
   *solution :* Le nombre de combinaisons est $2^4$ (pour chaque objet, il y a deux choix : le prendre ou non). Il lui faut donc $2^4 \times 5$, soit $10 \times 8$, c'est-à-dire $80s$ pour se décider. Avec quatre objets c'est envisageable, à partir d'une dizaine cela ne le serait plus du tout.
3. *Le voleur est informaticien :* Il utilise un algorithme de programmation dynamique pour calculer la solution optimale. Traiter la suite du problème...

## 2 Notations

On considére un sac à dos dont la capacité est notée $C$. On considère $n \in \mathbb{N}$ et $n$ objets, dont on note $\rho_0, ..., \rho_{n-1}$ les poids et $v_0, ..., v_{n-1}$ les valeurs.

Dans un premier temps, nous supposons que les poids et les valeurs sont des entiers strictement positifs. Dans les programmes, ces valeurs seront rentrées dans deux tableaux `rho` et `v`.

Ainsi le but est de choisir en sous-ensemble $X$ de $[\![0, n[\![$ tel que $\sum_{k \in X} \rho_k \leqslant C$ et pour lequel la valeur totale $\sum_{k \in X} v_k$ soit maximale.

## 3 Algorithme glouton

L'algorithme glouton est en réalité un peu plus fin que de prendre en priorité les objets ayant la plus grande valeur. Il s'agit de prendre en priorité les objets ayant le meilleur rapport valeur/poids. Pour tout $i \in [\![0, n[\![$, on notera $\alpha_i = \frac{v_i}{\rho_i}$ ce rapport.

1. Écrire une fonction `rapport_valeur_poids` prenant `rho` et `v` et renvoyant le tableau $[\alpha_0, \alpha_1, ..., \alpha_{n-1}]$.
   *solution :*

```
1
10 def rapport_valeur_poids(v,  ):
11     =[]
12     for i in range(len(v)):
13         .append(v[i ]/[i])
14     return
```

2. Écrire une fonction `meilleurObjetRestant` qui prend en entré le tableau `alpha` des $\alpha_i$ ainsi que le tableau `pris` et qui renvoie le numéro l'objet de meilleur rapport valeur/poids parmis les objets pas encore placés dans le sac à dos.
   Si tous les objets sont pris, on renverra $-1$.
   *solution :*

```
1
18 def meilleur_objet_restant(pris, ):
19     maxi=0 # Ici ça marche car les élément de  sont >0
20     res = -1
21     for i in range(len()):
22         if not pris[i] and [i] > maxi:
23             maxi =[i]
24             res = i
25     return res
```

3. En déduire la fonction finale. Elle renverra la liste des objets pris, ainsi que la valeur totale de ces objets.

   *solution :*

```
1
29 def glouton(v, , c):
30     n = len(v)
31       = rapport_valeur_poids (v, )
32     pris = [ False for i in range (0,n)]
33     res = []
34     c_restant = c
35     fini = False
36
37     while not fini :
38         i = meilleur_objet_restant(pris, )
39         if i != -1 and c_restant >= [i]:
40             res.append(i)
41             pris[i] = True
42             c_restant -= [i]
43         else:# Plus d'objet, ou l'objet restant est trop lourd.
44             fini = True
45
46     return res
```

4. Justifier la terminaison de votre programme.
   *solution :* Variant de boucle : le nombre d'objets non pris. C'est un entier positif, et à chaque itération de la boucle (sauf la dernière) il diminue de 1.

   On pourrait aussi prendre la capacité restante du sac.

5. Calculer la complexité de cette fonction en fonction du nombre d'objets $n$. On comptera le nombre de comparaisons entre éléments du tableau `alpha`.
   *solution :* L'appel à `rapport_poids_valeur` n'utilise pas de comparaisons (si on veut sa complexité, prendre par exemple les divisions : il y en a $n$. Ceci est de toute façon négligeable devant la complexité de ce qui suit).

   Ensuite la boucle principale est exécutée $n$ fois. À chaque itération, elle utilise `meilleurObjetRestant` qui coûte $n$ comparaisons.

   Au total, nous avons $n^2$ comparaisons.

6. M. Z teste son programme avec les données de **??**, et ce dernier lui indique de prendre le collier et le Cormen. Ce qui est quelque peu idiot puisqu'il y a encore la place pour la fourrure de glomorphe à rayures. Si votre fonction a le même comportement, expliquez celui-ci et corrigez celle-la.
   *solution :* Une fois pris le collier et le Cormen, le meilleur rapport valeur poids dans les objets restants va au candélabre. Mais comme il ne rentre pas dans le sac, le programme s'arrête.

   Pour le corriger, une possibilité est de modifier la fonction `meilleurObjetRestant` pour qu'elle ne prenne en compte que les objets qui rentrent dans le sac. Ceci nécessite de lui passer en argument la capacité restante et le tableau des poids.

```
1
50 def meilleur_objet_restantMieux(pris, , , c_restant): # Changement ici
51     maxi=0
52     res = -1
```

```
    for i in range(len ()):
        if not pris[i] and  [i] > maxi and  [i]<= c_restant : # Changement ici
            maxi =[i]
            res = i
    return res

def gloutonMieux(v,  , c):
    n = len(v)
      = rapport_valeur_poids (v,  )
    pris = [ False for i in range (0,n)]
    res = []
    c_restant = c
    fini = False

    while not fini :
        i = meilleur_objet_restantMieux(pris,  ,  , c_restant )# Changement ici
        if i != -1 and c_restant >=  [i]:
            res.append(i)
            pris[i] = True
            c_restant -=  [i]
        else:
            fini = True

    return res
```

7. Au moyen d'un tri on peut rendre cette fonction plus efficace. Expliquer comment, et indiquer la nouvelle complexité.
   *solution :* L'idée est de trier les objets par ordre décroissant de rapport valeur/poids. Il suffira alors de les prendre les un après les autres tant qu'il y a de la place dans le sac. Ceci pourra être fait en une seule boucle « for », donc en $O(n)$ comparaisons.
   On rajoute à ceci le coût du tri : cela fait $O(n \log n) + O(n)$ soit $O(n \log n)$ opérations.
   Détail technique : il ne suffit pas de trier le tableau `alpha` car alors on ignorerait à quel objet correspond chaque rapport. On peut d'abord créer un tableau de couples (rapport valeur/poids de l'objet i, i ). Comme Python, lorsqu'il compare deux couples, compare d'abord la première composante, la fonction **`sorted`** triera ce tableau selon le rapport valeur/poids.

```

def glouton_rapide(v,  , c):
    n=len(v)
      = [ (v[i]/  [i], i) for i in range(n) ]
     _trié = sorted ()

    c_restant = c
    res = []
    for k in range(n-1,-1,-1): # Lecture à l'envers puisque les objets les plus inté
    ↪ ressants sont à la fin du tableau
        _, i  =_trié[k]
        if  [i] <= c_restant:
            c_restant -=  [i]
            res.append(i)
    return res
```

8. Trouver un exemple ou l'algorithme glouton ne fournit pas la solution optimale. *Indication :* Le fait d'utiliser le rapport valeur/poids au lieu de simplement la valeur fait que l'exemple de **??** n'est plus un exemple où l'algorithme glouton n'est pas optimal.
   On peut par exemple prendre un cas à dos de capacité 8, et trois objets de poids 5, 4, et 4. Faire en sorte que la solution optimale soit de prenddre les deux objets de poids 4, mais que l'algorithme glouton choisisse l'objet de poids 5.

   *solution :* Prendre `v = [11, 8, 8]` et `rho = [5, 4, 4]`. Le premier objet a le meilleur rapport valeur/poids, il est donc choisi en premier. Mais il ne reste alors pas de place pour les autres, et la valeur totale du sac à dos est 11.
   Alors que la solution de prendre les deux autres objets permettait une valeur du sac de 16.

   Ou alors plus simple : faire en sorte que l'objet avec le meilleur rapport n'entre pas dans le sac. Alors le programme s'arrêtera immédiatement avec un sac vide !

# 4 Programmation dynamique

Pour tout $C \in \mathbb{N}$, et tout $i \in [\![0, n]\!]$, on pose $V(C, i)$ la valeur maximale des objets qu'on peut mettre dans un sac de capacité $C$ en choisissant uniquement des objets parmi les $i$ premiers (soit dans $[\![0, i[\![$). Si $C \leqslant 0$, nous convenons que $V(C, i) = 0$.

1. Que vaut $V(C, i)$ lorsque $C = 0$ ou $i = 0$ ?
   *solution :* Lorsque $C = 0$ on ne peut mettre aucun objet dans le sac (les poids sont supposés strictement positifs) donc $V(C, i) = 0$.

   Lorsque $i = 0$, l'ensemble $[\![0, i[\![$ des objets possibles est vide, donc $V(C, i) = 0$.

2. Démontrer que pour tout $C \in \mathbb{N}$ et $i \in [\![0, n[\![$,

$$V(C, i+1) = \max\left(V(C, i),\ V(C - \rho_i, i) + v_i\right)$$

   *solution :* Soit $C \in \mathbb{N}^*$ et $i \in [\![0, n]\!]$. Pour remplir notre sac à dos de capacité $C$ avec des objets de $[\![0, i+1[\![$ , il y a deux choix :

   - Prendre l'objet $i$. La capacité restante est alors $C - \rho_i$, et la valeur maximale qu'on peut y rajouter, sachant qu'il nous reste les objets $[\![0, i[\![$, est $V(C - \rho_i, i-1)$. Ainsi notre sac à dos aura une valeur totale de $V(C - \rho_i, i-1) + v_i$.
   - Ne pas prendre l'objet $i$. La valeur maximale possible est alors $V(C, i-1)$.

   La solution optimale est la meilleure de ces deux là, d'où la formule.

À présent, soit $C_0 \in \mathbb{N}$. La stratégie est de créer une matrice `V` de format $(C_0 + 1, n + 1)$ telle que pour tout $(c, i) \in [\![0, c]\!] \times [\![0, n]\!]$, `V[c][i]` contiendra une fois l'algorithme fini $V(c, i)$.

3. Écrire une fonction prenant $C_0$, $v$, *rho* et renvoyant la valeur maximale des objets qu'on peut mettre dans un sac de capacité $C_0$. L'algorithme utilisé consistera à créer puis remplir la matrice `V` grâce à la formule de la question précédente. *Indication :* Suivre exactement la formule ci-dessus. Prendre en particulier garde aux valeurs de $c$ et de $i$ pour lesquelles elle est valide.

   *solution :*

```
1
106 def sacÀDosDyna(c0, v,  ):
107     n=len ()
108     V= nouvelleMatrice(c0+1, n+1, 0)
109
110     for c in range(1,c0+1): # la ligne pour c==0 contient déjà des 0, qui est la bonne
        ↪ valeur.
111         for i in range(0,n):
112             V[c][i+1] = max( V[c][i], V[c -[i]][i] + v[i] )
113     # Notez à quel point on suit exactement la formule démontrée à la question précédente,
        ↪  notamment dans les bornes des boucles.
114
115     return V[c0][n]
```

4. Calculer la complexité de cette fonction. Comparer cette complexité à celle de la méthode naïve qui consiste à essayer toutes les combinaisons possibles d'objets.
   *solution :* Comptons par exemple le nombre de max utilisés. Il y en a $C0 \times n$.

   Pour la méthode naïve : le nombre d'ensembles d'objets possibles est $2^n$ (nombre de sous-ensembles de $[\![0, n]\!]$). Il aurait donc fallu $2^n - 1$ comparaisons pour obtenir la valeur maximale, c'est beaucoup plus que par la méthode proposée ci-dessus.

5. **Reconstruction de la solution optimale :** Le programme précédent ne calcule que la valeur maximale des objets qu'on peut mettre dans le sa à dos, et non la liste de ces objets. On peut employer deux méthodes pour obtenir cette dernière. La première consiste modifier le programme précédent pour remplir simultanément à la matrice `V` une matrice `listesObjets` telle que pour tout $c, i \in [\![0, C_0]\!] \times [\![0, n]\!]$, `listesObjets[c][i]` contiendra la liste des objets, choisis dans $[\![0, i[\![$, pour remplir au mieux un sac de capacité $c$. La seconde consiste à retrouver cette liste après coup en lisant le tableau `V` obtenu. L'idée est que si $V(c, i+1) = V(c, i)$ cela signifie que l'objet $i$ n'a pas été pris. Et réciproquement, si $V(c, i+1) = V(c - \rho_i, i) + V_i$ c'est que l'objet $i$ a été pris.
   Programmer une fonction prenant en entrée la matrice `V` remplie et renvoyant la liste des objets à prendre.
   *solution :*

```
1
148 def reconstruction(V,  , v):
149     res=[]
150     c=len(V)-1
151     i=len(V[0])-1 # c,i : case actuelle dans V
152     res=[]
153 
154     while c>0 and i >0 :
155         if V[c][i] == V[c][i-1] : #on n'a pas pris l'objet i-1
156             i=i-1
157         elif V[c][i] == V[c -[i-1]][i-1] + v[i-1] : # on a pris l'objet i-1
158             res.append(i-1)
159             c=c -[i-1]
160             i=i-1
161     return res
```

# 5 Mémoïzation

1. La méthode dynamique précédente (bottom-up) conduit-elle à calculer des valeurs inutiles de $V$?
   *solution :* Oui, et même beaucoup. Le premier exemple venu permet de s'en rendre compte. Prenont l'exemple de la partie ??.
2. Écrire une méthode « de haut en bas » basée sur une fonction récursive optimisée par mémoïzation.
   *solution :*

```
1
120 def sac_à_dos_mémo(c0, v,  ):
121     n = len(v)
122     V = nouvelle_matrice(c0+1, n+1, -1) # -1 signifiera « case pas encore remplie »
123 
124     # Initialisation : mettre des 0 dans la première ligne et la première colonne
125     for i in range(n+1):
126         V[0][i]=0
127     for c in range(c0+1):
128         V[c][0]=0
129 
130     def aux(c, i):
131         """ Renvoie V(c,i) et remplit la case correspondante de V si elle ne l'était pas
    ↪ encore. """
132         if c<=0:
133             return 0
134         elif V[c][i] != -1:
135              return V[c][i]
136         else:
137             res = max( aux(c,i-1), aux(c -[i-1], i-1) + v[i-1] ) # Attention, on utilise
    ↪ la formule précédente pour i-1 au lieu de i
138             V[c][i] = res
139             return res
140 
141 
142     # Résultat final
143     return aux(c0, n), V
```

# 6 Avec des flottants

Supposons que les poids ne sont plus des entiers. Laquelle des méthode précédentes s'adapte le mieux ? La programmer. Étudier la complexité.

*solution :* Déjà, les valeurs de $V$ ne sont plus indicées par des entiers : on va donc utiliser un dictionnaire pour les enregistrer. On ne pourra alors pas remplir ce dictionnaire par des boucles : il faut utiliser la méthode « de haut en bas ».

Cependant, compte tenu des erreurs d'arrondi, il se pourrait tout à fait que le programme ne reconnaisse pas qu'une valeur a déjà été calculée. Ainsi la complexité risque d'être mauvaise. Une solution serait de tronquer tous les flottants, quitte a perdre un peu de précision.