Projet n°03 - le problème du sac à dos

On étudie dans ce projet un problème très connu d'algorithmique : le problème du sac à dos. On dispose d'une famille de n objets, numérotés k, $k \in [\![0, n-1]\!]$. Chaque objet k possède deux caractéristiques : un poids p_k (en kilogramme par exemple, même si on devrait parler de masse ici...) et une valeur v_k (en euros par exemple). On suppose, ce qui sera important par la suite, que les poids et les valeurs sont des entiers strictement positifs.

On dispose également d'un sac à dos, qui peut emporter un poids maximal P, que l'on suppose également entier. Le but du problème est le suivant : *quel est la valeur maximale que l'on peut emporter dans le sac à dos en respectant le poids maximal* (autrement dit que la somme des poids des objets emportés reste inférieure ou égale à P), condition que l'on appellera C_0.

Bien sûr, il a deux situations où le problème est trivial : si tous les objets ont un poids strictement supérieur à P, cas où on ne peut prendre aucun objet ; et si P est supérieur ou égal à la somme des p_k, où on peut prendre tous les objets. On s'intéresse donc uniquement à la situation où :

$$P \in [\![\min(p_k), \sum_{k=0}^{n-1} p_k [\![$$

On suppose également que n est suffisamment grand pour évaluer des complexités asymptotiques.

On va s'intéresse à trois algorithmes : l'algorithme dit *naïf*, l'algorithme dit *glouton* et enfin l'algorithme optimal, issu de la *programmation dynamique*.

Exercice 1 Implémentation générale

Informatiquement, chaque objet sera donc décrit par un tuple `o` tel que `o[0]` corresponde à son poids, `o[1]` à sa valeur et `o[2]` son numéro (ceci, d'apparence inutile, sera utile pour les algorithmes). L'ensemble des objets sera donc une liste `L_objets`, contenant ces divers tuples. Par exemple :

```
L_objets=[(4,2,0),(3,3,1),(2,5,2)]
```

décrit trois objets, numérotés de 0 à 2, et l'objet 0 par exemple présente un poids de 4kg et une valeur de 2€.

1.1 Si `k` désigne le numéro d'un objet, à quoi correspond `L_objets[k][1]` ?

1.2 Coder une fonction `valeur_choix(L_e:list[int],L_objets:list[tuple[int]])->int`, qui prend en argument une liste `L_e` de taille inférieure ou égale à n, contenant les indices des objets emportés, et retournant, à partir de `L_objets`, la valeur totale des objets emportés.

1.3 Coder une fonction `poids_choix(L_e:list[int],L_objets:list[tuple[int]])->int`, qui prend en argument une liste `L_e` de taille inférieure ou égale à n, contenant les indices des objets emportés, et retournant, à partir de `L_objets`, le poids total des objets emportés.

Exercice 2 Algorithme naïf

L'algorithme naïf consiste simplement à essayer l'intégralité des combinaisons, et de prendre parmi celles qui vérifient C_0 celle(s) qui présentent la valeur maximale.

2.1 Expliquer pourquoi le nombre de combinaisons est de 2^n

2.2 Donner un ordre de grandeur, en puissance de 10, pour $n = 1000$ objets.

2.3 En déduire pourquoi on ne va même pas s'embêter à implémenter l'algorithme naïf...

Exercice 3 Algorithme glouton

Le principe de l'algorithme glouton [1] est le suivant : on trie les objets par **ratio $\frac{v_k}{p_k}$ décroissant** : autrement dit, après tri, `L_objets[0]` contient l'objet le plus "rentable" en termes de valeur pondérale.

3.1 Coder une fonction `ratio(o:tuple[int])->float`, prenant en argument un objet décrit par un tuple à trois éléments, et retournant la valeur (flottante) du ratio v/p. Par exemple `ratio((4,2,0))` doit retourner `0.5`.

On rappelle le principe du *tri par insertion inline*, appliqué à une liste de nombres `L`, à insérer dans une liste `Lt` qui sera in fine triée par ordre croissant :

1. on insère chaque élément de `L` à la fin de `Lt`
2. on initie un nombre `j` à la valeur du dernier indice de `Lt`
3. tant `j`>0 que le `j`ème élément de `Lt` est plus petit que le `j-1`ème élément :
   (a) on les permute
   (b) on décrémente `j` de 1.

3.2 Coder une fonction `tri(L_objets:list[tuple[int]])->list`, utilisant le tri par insertion inline, retournant une liste triée par **ratio décroissant**. Par exemple, une fois que l'on a appliqué le code suivant :

```
L_objets=[(4,2,0),(3,3,1),(2,5,2)]
L_triee=tri(L_objets)
```

alors `L_triee` contient `[(2, 5, 2), (3, 3, 1), (4, 2, 0)]` et `L_objets` n'est pas modifiée.

L'algorithme glouton consiste alors en la méthode suivante :

1. on génère la liste `L_triee` et une liste vide `L_e`
2. pour chaque objet o de `L_triee`, on insère $o[2]$ (son numéro) dans `L_e` si C_0 est respectée **après son ajout**, sinon on passe à l'objet suivant.

3.3 Coder une fonction `SaD_glouton(L_objets:list[tuple[int]];P:int)->list[int],int,int` qui renvoie un tuple à trois éléments contenant :

- la liste des numéros des objets choisis selon cet algorithme, avec `P` le poids maximal emportable
- la valeur des objets emportés (on pourra utiliser `valeur_choix`)
- le poids des objets emportés (on pourra utiliser `poids_choix`)

3.4 Expliquer pourquoi c'est la méthode de tri qui impose la complexité asymptotique de cet algorithme.

3.5 A-t-on gagné par rapport à l'algorithme naïf (on pourra évaluer l'ordre de grandeur pour $n = 1000$).

1. Un algorithme est dit *glouton* si il procède de la méthode suivante : on avance pas à pas en faisant le meilleur choix pour le pas suivant, en ne prenant en compte que ce pas et non une vision globale du problème.

3.6 On s'intéresse à la collection d'objets suivante (volontairement simple) :
`L_objets=[(4,10,0),(5,11,1),(4,8,2),(15,30,3)]` et un poids maximal de 15.

Appliquer `SaD_glouton` à cette liste d'objets, puis évaluer à l'aide de la fonction `valeur_choix` la valeur de la collection choisie (on peut le faire de tête par ailleurs). Montrer que cet algorithme ne renvoie pas la solution optimale, qui serait de prendre l'objet 3 pour une valeur de 30€ (alors que l'on n'a choisi que des objets de meilleur ratio que cet objet). Ceci est le problème général des algorithmes gloutons : ils sont simples à mettre en œuvre, de complexité en général faible, mais ne garantissent pas de renvoyer la solution optimale.

Exercice 4 Algorithme optimisé : programmation dynamique

Le principe général de la programmation dynamique est de construire la solution optimale d'un problème de taille n en trouvant une relation de récurrence entre la solution optimale du même problème, mais de taille k et la solution optimale d'un problème de taille $k+1$. Si on connaît la solution optimale pour le problème de taille 1, alors on peut donc par récurrence trouver la solution optimale pour le problème de taille 2, etc., jusqu'à n.

On va donc désigner le problème suivant : on appelle $A_{i,j}$ le problème du sac à dos en n'utilisant que les objets numérotés de 0 à i ($i \in [\![0, n-1]\!]$), avec un poids maximal j ($j \in [\![0, P]\!]$ et donc $V_{i,j}$ désigne la valeur optimale que l'on peut emmener en n'utilisant que les objets de 0 à i sans dépasser le poids j. On a quelques résultats quasi-triviaux (auxquels il faut bien réfléchir avant de commencer) :

1. le problème du sac à dos "complet" correspond à $A_{n-1,P}$ et la valeur optimale à $V_{n-1,P}$
2. $\forall i \in [\![0, n-1]\!]$, $V_{i,0} = 0$. Ceci est également vrai pour des valeurs de j négatives.
3. $V_{0,j} = 0$ si $j < p_0$, sinon $V_{0,j} = v_0$, où p_0 et v_0 sont respectivement le poids et la valeur de l'objet 0.

La situation est donc décrite par deux indices, et on va se représenter cela sous la forme d'un tableau bidimensionnel, comme figure 1. On a déjà rempli la ligne 0 grâce au point 3. précédent, ainsi que la colonne 0 grâce au point 2. précédent. On cherche la valeur qui est en bas à droite de ce tableau.

On va utiliser pour cela les tableaux `numpy`, importé comme suit :

```
import numpy as np
```

et on utilisera la commande `A=np.zeros((nl,nc),np.uint64)`, où `nl` et `nc` désigneront le nombre de lignes et le nombre de colonnes, et `np.uint64` indique que les éléments de ce tableau seront des entiers non signés sur 64 bits.

4.1 Donner, en fonction des caractéristiques de `L_objets` et de `P` :
- la valeur de `nl`
- la valeur de `nc` (attention il y a un piège...)
- l'ordre de grandeur de la valeur maximale d'un élément que l'on peut représenter dans ce tableau (en puissance de 10) au vu du type choisi.

La seule chose dont on a besoin est la nomenclature suivante : `V[i,j]` désigne l'élément à la ligne `i` et à la colonne `j`, tant en lecture qu'en écriture. Les tableaux `numpy` sont des objets mutables, donc tout se passe très bien. Par exemple, écrire `V[4,2]=5` affecte la valeur 5 à l'élément situé à la ligne 4 et à la colonne 2.

Si on se souvient que le problème se ramène, pour chaque objet, à choisir si on le prend ou non, on doit donc répondre à la question suivante : est-il intéressant, pour une situation connue donnée, de rajouter l'objet i, connaissant les divers problèmes où on a utilisé les objets jusqu'à $i-1$? On a donc à chaque fois deux choix :
- si on ajoute l'objet i, alors on augmente le poids total du sac de p_i et sa valeur totale de v_i
- sinon, le poids totale reste inchangé, ainsi que la valeur totale.

Si on suppose que $V_{i-1,p}$ contient la valeur optimale, en n'utilisant que les objets de 0 à $i-1$, pour un poids maximal p, alors :

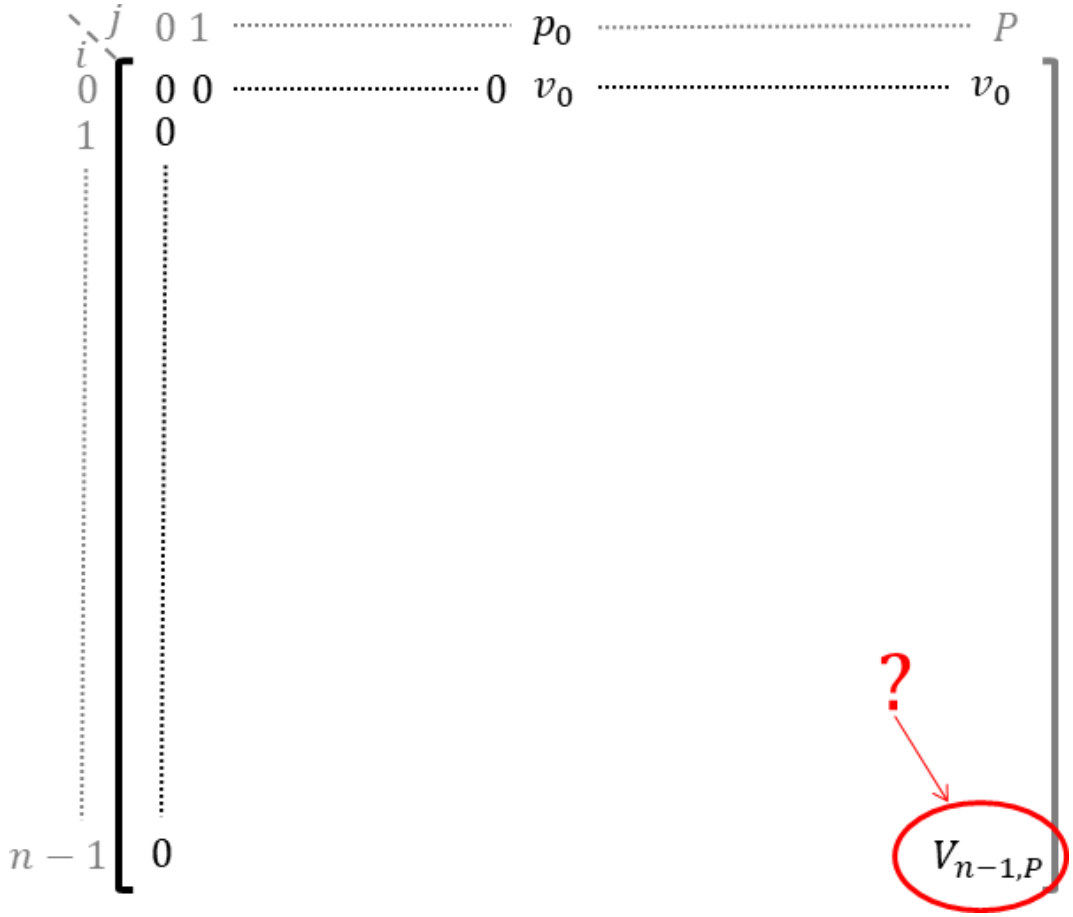

FIGURE 1 – Initialisation du problème du sac à dos

- si la solution optimale n'utilise pas l'objet i, $V_{i,p} = V_{i-1,p}$: on n'a pas augmenté de valeur
- si la solution optimale utilise l'objet, alors il faut tenir compte de l'ajout de poids : la solution optimale utilisant les i premiers objets pour un poids maximal p est issue de la solution optimale utilisant les $i-1$ premiers objets, mais pour un poids maximal de $p-p_i$ (à condition que cette différence soit positive). En effet, si la solution en utilisant l'objet i est optimale pour un poids maximal p, ceci signifie que la situation était également optimale à l'étape d'avant, et donc pour un poids maximal $p - p_i$.

Comme "solution optimale" signifie ici "valeur maximale", on en déduit qu'utiliser l'objet i est optimal si et seulement si $V_{i-1,p-p_i} + v_i$ (valeur maximale à $p - p_i$ à laquelle on rajoute v_i) est supérieure à $V_{i-1,p}$ (valeur du sac où on n'a rien changé). Ceci est schématisé figure 2

On en déduit la relation de récurrence suivante, où le premier cas traite les poids maximaux où il est impossible de mettre l'objet i car le sac ne peut pas le porter ($j < p_i$) :

$$\forall (i,j) \in [\![1, n-1]\!] \times [\![0, P]\!], V_{i,j} = \begin{cases} V_{i-1,j} & \text{si } j < p_i \\ \max\left(V_{i-1,j}, V_{i-1,j-p_i} + v_i\right) & \text{sinon} \end{cases}$$

Comme souvent en programmation dynamique, on n'obtient pas directement la liste des objets à emporter, mais uniquement la valeur maximale que l'on peut emmener. Il faut ensuite *reconstruire la solution*. Ceci sera abordé plus loin.

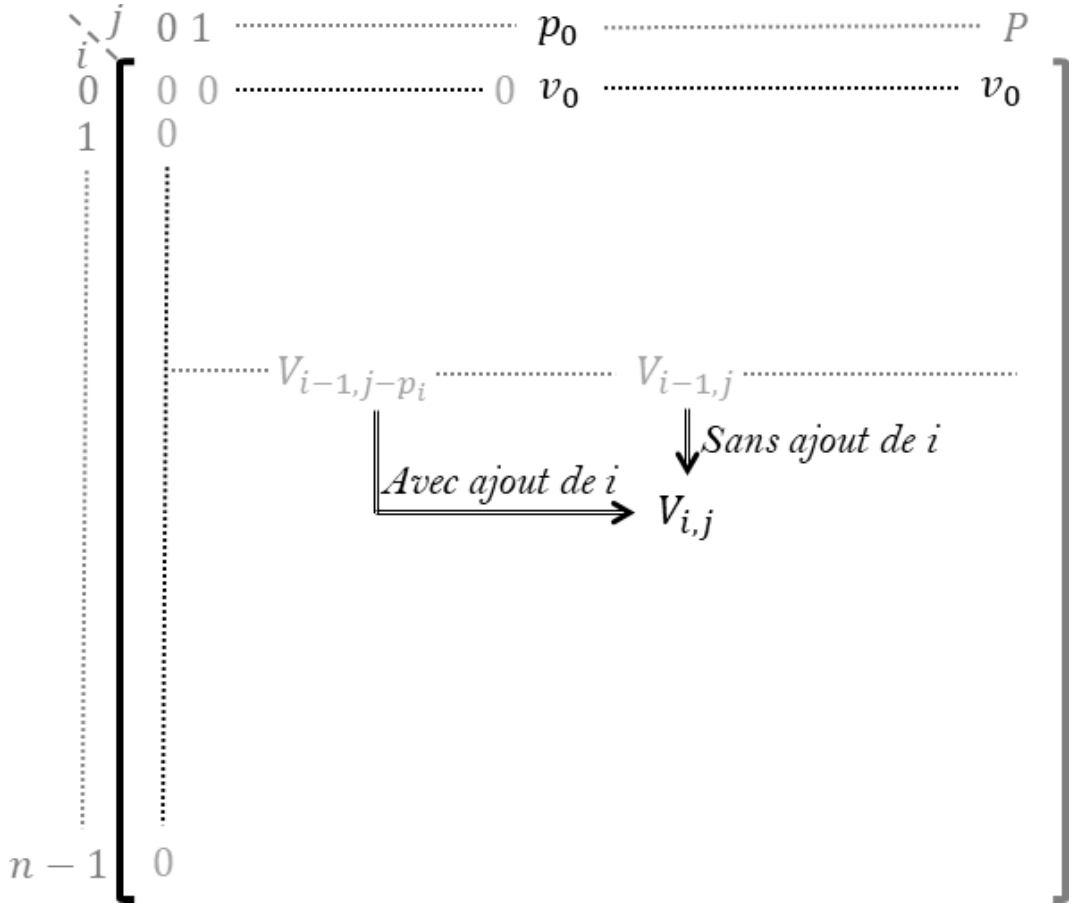

FIGURE 2 – Relation de récurrence du problème du sac à dos

On va donc fabriquer une fonction `SaD_pDyn` qui code cet algorithme, en complétant l'ossature fournie :

```
def SaD_pDyn(L_objets:list[tuple[int]],P:int)->(list[int],int,int):
    #initialisation
    V=np.zeros((...,...),np.uint64)
    for i in range(len(V)):
        o=L_objets[i]
        for j in range(len(V[i])):
            if i==0: #cas de base
                if ...:
                    V[i,j]=...
            else:
                if ... or ...:
                    V[i,j]=V[i-1,j]
                else:
                    V[i,j]=...

    print(V) #non obligatoire

    #reconstruction de la solution:
    L_e=[]
    j=...
    for i in range(len(V)-1,-1,-1):
        if i>0:
            if ...:#on a pris l'objet i
                L_e.append(i)
                j-=...
```

```
        else:
            if ...: #on a pris l'objet 0
                L_e.append(i)
    return L_e,valeur_choix(L_e,L_objets),poids_choix(L_e,L_objets)
```

4.2 Compléter les lignes de 1 à 15 traduisant la relation de récurrence et l'implémentation via un tableau `numpy`. On pourra remarquer que lar elation de récurrence se réécrit :

- si $j <_p i$ ou si $V_{i-1},j > V_{i-1,j-p_i} + v_i$, alors $V_{i,j} = V_{i-1,j}$
- sinon $V_{i,j} = V_{i-1,j-p_i} + v_i$

La reconstruction de la solution, c'est-à-dire la construction de la liste `L_e`, se fait de la façon suivante :

- on part du coin en bas à droite, qui contient la solution optimale ; pour cela, on initialise une variable `j` à `P`
- on parcourt les lignes de la dernière à la première.
- si $i > 0$ et si $v_{i,j} > v_{i-1,j}$, alors on a utilisé l'objet numéro i : on insère i dans `L_e`, et on réduit j du poids de i
- si $i == 0$, on regarde si $v_{0,j}$ est non nul. Si c'est le cas, on a utilisé l'objet 0, et on insère 0 dans `L_e`

4.3 Finir de compléter l'algorithme avec ces indications.

4.4 Évaluer, en fonction de n et P, la complexité asymptotique de cet algorithme, et comparer avec celle de l'algorithme glouton.

Exercice 5 Comparaison entre les deux algorithmes

On va pour finir essayer de comparer les deux algorithmes en utilisant le code suivant :

```
from random import randint,random
N=100 #nombre d'objets
L_objets2=[]
somme_p=0
for i in range(N):
    r=random()*0.2+5 #on contraint les objets a avoir un ration v/p
       [... similaire (entre 5 et 5.2 ici)
    p=randint(100,1000) #les poids sont variables
    v=int(r*p)
    L_objets2.append((p,v,i))
    somme_p+=p

P=somme_p//3 #on se limite à un tiers du poids total, ce qui enlève les
   [... cas triviaux et oblige à faire des bons choix.
print(SaD_glouton(L_objets2,P))
print(SaD_pDyn(L_objets2,P))
```

5.1 Expliquer ligne à ligne ce que fait ce code.

5.2 Exécuter ce programme et vérifier :

- que l'algorithme glouton prend nettement moins de temps
- que l'algorithme glouton ne retourne que rarement la solution optimale (même si on n'est jamais vraiment loin)