Le problème du sac à DOS

— The knapSack Problem —

Il s'agit ici d'un grand classique de l'algorithmique, connu depuis la fin du XIXème siècle, qui pose une question d'optimisation très simple :

« Étant donné un sac à dos pouvant contenir une capacité en kilos, et une collection d'objets (indexés par i) de valeur vi et de masse mi, comment choisir les objets à mettre dans le sac à dos pour obtenir la valeur maximale sans toutefois dépasser la capacité du sac à dos ? »

Ce problème formulé de façon un peu naïve s'applique à quantité de situations comme l'optimisation du remplissage des conteneurs, camions, mais aussi à l'optimisation des plans de découpe de meuble, pièce mécanique.

Un intérêt pédagogique de ce problème est la quantité de démarches possibles pour son approche et aussi le fait qu'on peut vouloir trouver vite une « bonne » solution ou chercher la garantie d'avoir la « meilleure » solution possible. La question de la complexité est alors posée.

Exemple :

Soit un sac de capacité 10 kilos et la collection d'objets suivante (à copier/coller) :
maxWeigth = 10
listeObjet = [(8, 10), (5, 15), (3, 1), (4, 3), (9, 12)] # liste de tuples objets —> (poids, valeur)

On donne ici des valeurs entières pour pouvoir aborder le problème « à la main » mais dans la suite on imagine que les valeurs et les poids sont des nombres flottants quelconques.

I Préparation : analyse du problème

a - Soit N le nombre d'objets, combien de « prises » d'objets sont possibles, indépendamment de la capacité du sac ? Proposer une complexité pour tester toutes les possibilités de remplissage.

b - Proposer la solution qui vous parait optimale dans l'exemple précédent. Avez-vous besoin de tester toutes les combinaisons d'objets ?

c - Python possède un module knapSack qui permet d'obtenir la solution optimale :

```
import knapsack
sol = knapsack.knapsack(weight, value).solve(maxWeigth)
print(« Solution témoin module knapSack : », sol)
```

Elle renvoie la valeur optimale et les indices des objets mis dans le sac.
On suppose que vous disposez des variables maxWeight et listeObjet de l'exemple :
- Construire la liste weight contenant uniquement les poids des objets [compréhension de liste].
- Construire la liste value contenant uniquement les valeurs des objets [compréhension de liste].
- Comparez la solution que vous aviez obtenue à la solution du module.

II Algorithme glouton

Une idée simple est de prendre toujours l'objet de plus grande valeur qui rentre en fonction de la capacité restante.

a - Ecrire une fonction **getMax**(mesObjets) qui prend en argument une liste d'objets et qui renvoie le tuple (iMax, w, vMax) des indice, poids et valeur de l'objet de plus grande valeur.

b - Ecrire une fonction **glouton**(listeObjet, maxWeight) qui prend en argument la liste des objets ainsi que la capacité, et qui renvoie le résultat du problème obtenu par le glouton sous la même forme que le module knapsack : (valeur totale, liste des indices des objets pris) —> (18, [1, 3])

Remarque : cette fonction utilisera la fonction **getMax** précédente.
Il peut être nécessaire de modifier la liste des objets, auquel cas à vous de travailler sur une **copie** de l'objet, mais la liste d'objets **listObjet ne doit pas être modifiée à la fin du processus.**

c - Quelle est la complexité de l'algorithme glouton en fonction du nombre N d'objets ?
Peut-on / doit-on qualifier cette complexité de Polynomiale ou de Non Polynomiale.

d - L'algorithme glouton trouve t-il la meilleure solution dans tous les cas ? Proposez une modification de la liste d'objet (créer **listeModif**) pour montrer que la solution de glouton peut ne pas être optimale. On comparera avec la solution donnée par le module knapsack pour cette liste.

e - Il est assez aisé de « feinter » l'algorithme glouton, il suffit de donner un petit poids aux 2ème et 3ème plus grande valeurs qui seraient par ailleurs proche de la première valeur de poids fort : le glouton va se jeter bêtement sur la 1ère ….
De fait une optimisation très simple permet de grandement l'améliorer : Il faut raisonner sur la densité valeur/poids plutôt que sur la valeur elle-même !
- Construire la liste **listeModif_density** qui contient les mêmes objets mais sous la forme : (poids, valeur/poids). On procédera par compréhension de liste.
- Montrer que la solution du glouton est beaucoup plus robuste que précédemment.
- Trouver un contre exemple pour montrer que toutefois le glouton ne trouvera pas nécessairement la solution optimale.

Rq : on se contentera de constater que les objets sélectionnés ne sont pas les mêmes
En revanche le calcul du gain avec la densité nécessiterais de modifier la fonction glouton, car la valeur prise vaut poids fois densité. Vous pouvez le faire de votre coté.

III Programmation dynamique

La programmation dynamique consiste pour résoudre un problème de taille N à s'appuyer sur les solutions aux problèmes de tailles inférieures préalablement obtenues.

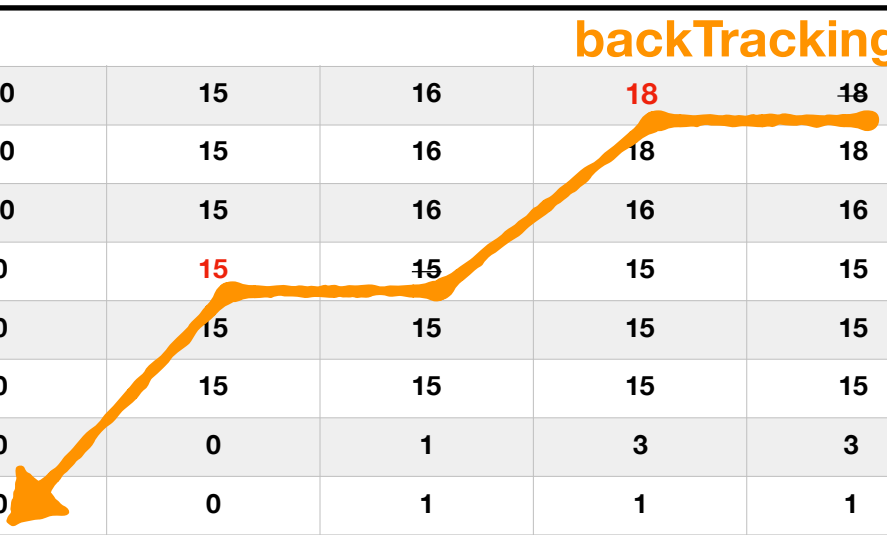
On utilise pour cela un tableau à 2 - dimensions dont :
- les les colonnes sont les différents objets du problème. Chacun est caractérisé par une valeur et un poids pour entrer dans le sac.
- les lignes (de bas en haut) sont tous les poids entiers possibles. On suppose que l'unité de mesure est adaptée à la précision désirée. On peut toujours représenter 15.343 kg par 15343g.

Les cases du tableau contiendront en (i, j) la valeur optimale si on a essayé d'introduire les i premiers objets et que je poids est inférieur ou égal à j.

On crée la variable **optimSelect** représentant le double tableau initialement vide
optimSelect = [[ ]]

Important :
- ce tableau sera donc une liste de colonne indexées par i (premier indice => colonne objet)
- chaque colonne sera une liste de poids croissants indexés par j (second indice => ligne poids)

Poids sac					
10	10	15	16	18	~~18~~
9	10	15	16	~~18~~	18
8	10	15	16	16	16
7	0	15	~~15~~	15	15
6	0	~~15~~	15	15	15
5	0	15	15	15	15
4	0	0	1	3	3
3	0	0	1	1	1
2	0	0	0	0	0
1	0	0	0	0	0
0	0	0	0	0	0
	Objet 1	Objet 2	Objet 3	Objet 4	Objet 5
Valeur	10	15	1	3	12
Poids	8	5	3	4	9



La méthode de programmation dynamique s'appuyant sur les résultats de taille de tableaux plus faibles, il faut au départ au moins avoir rempli la première colonne. De même la ligne du bas vaut nécessairement 0. Nous placerons systématiquement un zéro en début de colonne.

a - Construction et remplissage de la première colonne

Remplir de bas en haut la première colonne en fonction du poids résultant pour le sac :
- soit l'objet rentre pour ce poids => on indique sa valeur.
- soit il ne rentre pas encore => on indique 0 de valeur introduite.

b - Ecrire un « bout de code » réalisant automatiquement cette tâche dans le programme.
Attention : le premier objet correspondra en fait à l'indice i = 0.

Pour remplir le tableau complètement, on procède colonne par colonne.
Pour remplir une colonne on passe en revue les poids croissants et on compare 2 possibilités :
- Soit on ne prend pas l'objet de la colonne :
  Auquel cas la valeur dans le sac sera la valeur optimale déjà obtenue pour l'objet précédent au même poids. Il suffit de lire la case d'à coté.
- Soit on prend l'objet de la colonne (si il est possible de le prendre) :
  La valeur sera celle de l'objet à laquelle on ajoute la solution optimale de la colonne précédente mais **attention** : **pour un poids qui est celui de la ligne moins le poids de l'objet !**

On compare ainsi les 2 possibilités et on garde la plus opportune **Grreeeeed is GoOOod !**

c - Remplir le tableau à la main colonne par colonne. Prenez le temps de bien vous approprier l'algorithme dynamique sur cet exemple.

d - Ecrire un « bout de code » qui remplit automatiquement le tableau.
On procède par re-construction du tableau :
—> re-consruire une colonne initialisée avec le 0 de la première ligne
—> re-construire les lignes en appliquant l'algorithme dynamique
Rq : notez bien qu'il faut vérifier que l'objet peut rentrer dans le sac avant de l'y mettre.

e - Le gain optimal est évident si l'on a rempli le tableau. Que vaut-il sur notre exemple ?
Où est-il dans le tableau et pourquoi ? Peut-on le démontrer et par quel type de raisonnement ?

En pratique le gain max n'est pas le plus important, il faut surtout avoir la liste des objets à prendre pour réaliser ce gain max !!! Le tableau va nous y aider.

backTracking de la solution optimale.

Il suffit en effet de revenir sur nos pas à travers le tableau en partant de la solution optimale.
Pour chaque case sur notre chemin on applique l'algorithme suivant :

- Soit la valeur à gauche est identique => On avait pas pris l'objet de la colonne.
  La capacité du sac reste inchangée.

- Soit la valeur à gauche est plus faible => On avait pris l'objet de la colonne.
  Il faut retrancher son poids à la capacité restante,
  et on descend sur cette ligne du tableau.

- Soit la case vaut 0 => c'est fini, car il n'y aura plus rien à prendre pour cette capacité !

Attention : si le premier objet doit être pris, il n'y aura pas de case à gauche (pas de colonne) et la case ne vaudra pas 0 non plus ! Il faut traiter ce cas particulier à part.

f - Tracer la trajectoire du backTracking dans le tableau en entourant les objets mis dans le sac et en barrant ceux que l'on a écartés. A nouveau prenez le temps de bien vous approprier l'algorithme du backTracking.

g - Ecrire un « bout de code » qui réalise cet algorithme et place des 0 et des 1 dans une liste. On introduira une liste **selectionObjets** qui vaut 0 si l'objet n'est pas pris et 1 si il est pris. Dans la foulée stocker dans une autre liste **selected** les indices des objets sélectionnés dans l'ordre croissant pour avoir un affichage similaire à celui du module knapSack.

Attention :
Penser à traiter le cas particulier où le backTracking prend fin sur le premier objet en i = 0.

h - Afficher le résultat de votre programme à la manière du module knapSack.

i - On peut chronométrer nos différentes résolutions pour voir laquelle est la plus rapide. On utilisera pour cela la module time :

```
from time import clock
start=clock()
#programme à chronométrer
stop=clock()
print("durée :", stop-start)
```

j - Pour comparer enfin les performances des différents algorithmes rencontrés, on utilisera un générateur de listes d'objets aléatoires : **commenter les lignes suivantes**.

```
import random as rnd   #C1

poidsMax  =  10000      #C2
valeurMax =  100.0      #C3
maxWeight = poidsMax

N = 25  #augmenter ce nombre très progressivement.
listeObjet = [ ( rnd.randint(0, poidsMax), rnd.random()*valeurMax ) for k in range(N)]
#C4  —> expliquer la commande précédente.
```

Rq : la liste est aléatoire, mais **on doit toujours comparer les algorithmes pour une même liste**. Vous noterez que la durée et le classement des algorithmes dépend beaucoup de la liste.

IV Algorithme arborescent : arbre binaire

L'idée est ici de tester toutes les possibilités mais à la manière d'un arbre « binaire » ou « dichotomique ». Soit la liste des objets i de 0 à N -1 dans un ordre quelconque, pour chaque objet : **on prend ou on ne prend pas**.

a - Dessiner l'arbre des possibilités avec l'exemple de départ pour 5 objets. Le premier objet est la racine : on prend => part à gauche on prend pas => part à droite. On obtient ainsi 2 noeuds pour le second objet qui se subdivise chacun en 2 etc…
Combien y a t'il de « feuilles » à la fin du processus ? En déduire la complexité maximale.

On veut programmer un parcours d'arbre de façon récursive. On passe en revue tous les noeuds de l'arbre et une fois arrivé en bas (aux feuilles donc…) on regarde la valeur dans le sac, que l'on compare à la meilleur solution trouvée à ce stade. On peut également s'apercevoir en chemin que l'on a déjà dépasser la capacité du sac, ce qui signifie qu'il est inutile de poursuivre la branche. On propose donc l'algorithme suivant :

b - Initialiser une solution optimale **bestVal** qui vaudra initialement 0 et la liste de prise des objets **listePrise** [0 si l'objet n'est pas pris et 1 s'il est pris]. Initialement listePrise contiendra N fois 0 et de même **bestListe** qui gardera en mémoire la meilleure collection trouvée à l'instant t.

c - On écrit une fonction récursive **prendObjet(mesObjets, i, P, V, bestVal, bestListe)** qui va tester les 2 possibilités : prendre ou pas le i-ème objet, sachant que le sac contient déjà un poids P et une valeur totale V, et en connaissant la liste des objets ainsi que la meilleur valeur trouvée à ce stade et la collection d'objets correspondant. Cette fonction doit :

- Comparer la valeur trouvée à la valeur optimale si l'on est sur une feuille [plus d'objet à prendre]. Si la valeur trouvée est meilleure, modifier bestVal et bestListe et retourner le résultat. C'est la **condition de terminaison**.
- Récupérer les caractéristiques de l'objet sinon.
- Tester le cas où l'on ne prend pas l'objet à l'aide d'un appel récursif **prendObjet** adapté.
- Tester le cas où l'on prend l'objet à l'aide d'un appel récursif **prendObjet** adapté. (Dans ces 2 cas mettre à jour **listePrise** le long de la branche en cours)

Dans tous les cas **la fonction prendObjet renvoie le tuple (bestVal, bestListe)**. Ceci permet de garder « à jour » les valeurs optimale et la collection associée tout au long du parcours de l'arbre.

L'appel initial à cette fonction se fera pour i=0, P=0, V=0, bestVal=0 et une liste bestListe de 0.

Conseil : utiliser le diagramme de l'arbre pour vous guider dans la rédaction de votre code.

d - Ecrire une fonction **getListIndices**(liste10) qui transforme la liste de 0 et de 1 en une liste des indices des objets pris. Exemple : [0,1,0,1,0]—> [1, 3]

e - Réaliser le chronométrage et les affichages pour la partie IV comme pour les précédentes.

Optimisation par un tri en densité

A ce stade les objets sont pris dans un ordre aléatoire. Pourtant nous avons vu pour la méthode glouton qu'il est souvent plus efficace de regarder les objets non en fonction de leur valeur mais en fonction de leur densité de valeur : valeur par unité de masse.

En effet si on re-construit notre arbre avec des objets pris en densité décroissante, on comprend vite qu'à chaque fois qu'un objet est ajouté, la densité moyenne dans le sac diminue !
Un critère simple permet alors de couper les branches plus près de la racine :
si la densité actuelle du sac fois la capacité maximale du sac devient inférieure à la valeur optimale actuelle (sachant que la densité va diminuer en descendant) on est sûr de ne jamais plus atteindre une valeur meilleure que la valeur optimale actuelle ! Il est donc inutile de suivre cette branche plus longtemps : quick !

On donne les éléments de syntaxe suivants qui permettent de trier une liste de tuples selon le n-ième élément du tuple :

```
import operator
listeTuple.sort(key=operator.itemgetter(n))  #LA formule magique ....
```

La commande suivante renverse l'ordre :

```
listeTuple.reverse()
```

f - Ecrire une fonction **triTuple**(mesObjets) qui renvoie les mêmes objets mais triés par densités décroissantes. Vous devrez créer une liste intermédiaire ayant (poids, valeur, densité), **trier dans le bon ordre** les tuples, puis ne renvoyer que les tuples (poids, valeur).

g - Copier/coller votre fonction **prendObjet** et la modifier en **prendObjetDense**, fonction qui exploite le fait que les noeuds sont rencontrés par densité décroissante pour couper les branches le plus tôt possible.

h - Refaites les chronométrages nécessaires pour différentes tailles de listes et conclure.

FIN