I Présentation du problème

Énoncé

L'énoncé de ce problème est assez simple :

- On dispose d'une collection d'objets ayant chacun une masse et une valeur, ainsi que d'un sac à dos qui a une contenance maximale (en kg), inférieure à la masse totale des objets.
- La question est : quelle est la sélection optimale d'objets à placer dans le sac à dos pour maximiser la valeur transportée ?

Ce problème est un problème classique d'optimisation "combinatoire"

Numérisation

Dans la suite, le sac à dos sera représenté par sa capacité M_{max} qui sera un entier, et les objets seront chacun représentés par un couple `(m, v)` d'entiers qui seront dans cet ordre la masse et la valeur.
Par exemple

```
M_max = 15
objets = [(2, 1), (5, 2), (7, 3), (9, 10), (12, 7)]
```

Un remplissage sera alors une liste d'objets dont la somme des masses est inférieure ou égale à M_{max} et on cherche un remplissage de valeur totale maximale.

Exercice 1
Quelle est la solution optimale pour cet exemple ?

II Approche gloutonne

Une première approche de ce problème consiste à introduire **l'efficacité** de chaque objet que l'on définit par $\frac{m}{p}$ et qui s'exprime en valeur/kg.

Exercice 2
Compléter la fonction `efficacite` présente dans la feuille `tp5.py`

L'algorithme est maintenant simple : on choisit les objets ayant la plus grande efficacité pour remplir le sac, tout en gardant la condition de masse satisfaite. Plus précisément :

1. Trier la liste des objets par efficacité décroissante.
2. Pour chaque objet dans la liste triée, l'ajouter au remplissage s'il ne fait pas dépasser la masse maximale autorisée.

Nous allons utiliser la méthode `sort` des listes python qui permet de trier des listes, étudions attentivement son utilisation

```
help([].sort)
```

Exercice 3
Lire attentivement le teste affiché après l'exécution de la commande précédente. on voit apparaître deux argument optionnels (sous la forme `nom=valeur par défaut`).
Étudier ensuite les exemples suivants (qui sont présents sous forme de commentaires dans `tp5.py`).
Vous pouvez ensuite voir le contenu des listes `L1` et `L2` dans l'onglet "Variable explorer" au dessus de la console.

```
def f1(x):
    return x**2

L1 = [-4, -2, 1, 3, 5]
L1.sort(key=f1)

def f2(c):
    return c[0] + c[1]

L2 = [(2, 1), (5, 2), (7, 3), (9, 10), (12, 7)]
L2.sort(key=f2)
```

```
L1.sort(reverse=True)
```

Exercice 4
Compléter la fonction `glouton` qui implémente l'algorithme décrit précédemment.
Pourquoi est-ce un algorithme glouton ? La solution retournée sur notre exemple est-elle optimale ?

Exercice 5
En admettant que le tri des listes python est un tri rapide (complexité $O(n \ln n)$ où n est la longueur de la liste à trier), quelle est la complexité de l'algorithme glouton mis en uvre ici en fonction du nombre d'objet n ?

III Force brute

Une deuxième méthode, exacte cette fois, est de considérer tous les remplissages possible et conserver celui qui atteint la valeur maximale.

Le fichier `tp5.py` contient une fonction `sous_listes` que vous n'avez pas à comprendre mais qui permet d'itérer sur toutes les sous-listes non vide d'une liste `L` donnée. Plus précisément, sa valeur de retour est un objet que l'on peut placer après le mot clé `in` dans une boucle `for`

```
L = [1,2,3]
for sL in sous_listes(L):
    print(sL)
```

Exercice 6
Dans le cas d'une liste `L` de longueur n, combien de sous-liste non vide possède `L` ? Vous pouvez expérimenter dans la console pour avoir une intuition du résultat puis vous devez le justifier.

Exercice 7
Compléter la fonction `valeur` et la tester.

Exercice 8
Compléter la fonction `brutal` et retrouver le remplissage optimale pour notre exemple.

IV Programmation dynamique

Le principe général de la programmation dynamique est de construire une solution à un problème (le sac à dos par exemple), à partir d'une solution à un problème plus "petit".
Nous avons déjà traité en cours un algorithme de tri en utilisant cette idée (tri fusion).

Application au problème du sac à dos.

Posons d'abord quelques notations.

- On note n le nombre d'objets dans la collection d'objet.
- On note m_i, v_i la masse et la valeur de l'objet i pour $i \in [\![1, n]\!]$.
- Pour $k \in [\![0, n]\!]$ et $m \in \mathbb{N}$, on note $f(k, m)$ la valeur maximale solution du problème du sac à dos composé des k premiers objets et de masse maximale m (qui n'est pas forcément la masse maximale autorisée par notre sac à dos).

On a alors :

$$\begin{cases} f(0, m) = 0 \text{ pour tout } m \\ f(k, m) = f(k-1, m) \text{ si } m_k > m \\ f(k, m) = \max(f(k-1, m),\ f(k-1, c-m_k) + v_k) \text{ si } m_k \leqslant m \end{cases}$$

Implémentations

Exercice 9
Traduire les formules précédentes dans une fonction récursive, `valeur_rec`. On prendra garde au fait que l'entier k de la formule n'est pas un indice dans la liste des objets.
En déduire une implémentation en une ligne de la fonction `valeur_max` qui calcule la valeur optimale solution du problème du sac à dos.

En modifiant légèrement le code précédent, on peut obtenir en plus de la valeur maximale, la liste des objets à choisir, mais nous ne le ferons pas aujourd'hui.
Le code précédent n'est pas efficace, car on effectue plusieurs fois le même appel récursif. On peut le rendre plus efficace en mémorisant les résultats intermédiaires, et c'est une technique classique de programmation dynamique.

Exercice 10
Au lieu de considérer f comme une fonction, on crée une matrice de taille $(n+1) \times (\mathrm{M}_{\max}+1)$ dont le coefficient d'indices (on parle d'indice dans la matrice python) k et m vaut $f(k,m)$.
La définition de f permet de construire la matrice ligne par ligne en initialisant la première ligne à 0.
Implémenter cette construction dans la fonction `sac_memoire` qui retourne la matrice construite.

Exercice 11
Comment déduire de la matrice précédente un remplissage optimal ? Quelle est la complexité de cette approche ?

Pour aller plus loin

Proposer une implémentation de la fonction `sous_liste`.