Exercice : Le problème du sac à dos

Le **problème du sac à dos**, parfois noté **KP** (*Knapsack Problem*) est un problème d'optimisation combinatoire. Il consiste à trouver la combinaison d'objets la plus précieuse à inclure dans un sac à dos, étant donné un ensemble d'objets de poids et de valeurs variables.

L'objectif du problème du sac à dos est de maximiser la valeur des objets pouvant être placés dans le sac à dos, sous contrainte : le poids total des objets ne doit pas dépasser la capacité du sac à dos. Ce problème est considéré comme NP-difficile, ce qui signifie qu'il est difficile à résoudre, en particulier pour de grands ensembles d'objets.

Dans le cadre de cet exercice nous utiliserons en entrées :

- la capacité c du sac
- un ensemble d'objets de poids $poids_1$, $poids_2$, …, $poids_n$ et valeurs , v_1,v_2…, v_n

La résolution du problème permettra d'obtenir la valeur maximale que l'on peut mettre dans un sac de capacité c (c correspond donc au poids total maximal que l'on peut emporter dans le sac).
L'objectif de cet exercice est de comparer des approches par algorithmes gloutons à une approche par programmation dynamique.

Algorithmes gloutons

Un algorithme glouton consiste à ajouter des objets un par un au sac, en choisissant à chaque étape l'objet qui a l'air le plus intéressant, si son poids n'excède pas la capacité restante du sac. Suivant l'ordre (et par conséquent les critères) dans lequel on choisit les objets, on obtient des algorithmes gloutons différents.

1. Écrire une **fonction glouton(c, poids, val)** qui renvoie la valeur totale des objets choisis par l'algorithme glouton, en considérant les objets dans l'ordre donné par les listes poids et val (on regarde d'abord l'objet de poids poids[0] et valeur val[0], puis l'objet de poids poids[1] et valeur val[1]…).
Exemple :

```
glouton(10, [5, 3, 6], [4, 4, 6])    # renvoie 8
```

Le résultat est-il optimal ?

Tri des objets

2. Écrire une **fonction combine(L1, L2)** qui renvoie la liste des couples (L1[i], L2[i]). On suppose que les listes L1 et L2 sont de même longueur.
Exemple :

```
print ( combine([1, 2, 3], [4, 5, 6])  ) # [(1, 4), (2, 5), (3, 6)]
```

3. Écrire une **fonction split(L)** telle que si L est une liste de couples, split(L) renvoie deux listes L1 et L2 où L1 contient les premiers éléments des couples de L et L2 les seconds éléments des couples de L.
Exemple :

```
split([(1, 4), (2, 5), (3, 6)])  # retourne([1, 2, 3], [4, 5, 6])
```

Si L est une liste, L.sort() trie L par ordre croissant.
L.sort(reverse=True) permet de trier par ordre décroissant.
Si L contient des couples, la liste est triée suivant le premier élément de chaque couple (ordre lexicographique).
Exemple :

```
L = [(1, 4), (7, 5), (3, 6)]
L.sort()  # L est maintenant triée suivant le 1er élément de chaque tuple [(1, 4), (3, 6), (7, 5)]
```

4. A l'aide des fonctions combine et split, définir une **fonction tri_poids(poids, val)** qui renvoie les listes poids2 et v2 obtenues à partir de poids et v en triant les poids par ordre croissant.
Exemple :

```
tri_poids([5, 3, 6], [42, 0, 2]) #([3, 5, 6], [0, 42, 2])
```

Stratégies gloutonnes

5. En déduire une **fonction glouton_poids(c, poids, val)** qui renvoie la valeur totale des objets choisis par l'algorithme glouton, en considérant les objets dans l'ordre de poids croissant. On cherche ici à maximiser le nombre d'objets emportés. On pourra réutiliser glouton.

```
glouton_poids(10, [5, 3, 6], [4, 4, 10])   #retourne 8
```

Est-ce que cet algorithme est toujours optimal ?

6. Écrire de même des **fonctions tri_valeur(poids, val) et glouton_valeur(c, poids, val)** qui renvoie la valeur totale des objets choisis par l'algorithme glouton, en considérant les objets dans l'ordre de valeur décroissante (en utilisant L.sort(reverse=True)).
Exemple :

```
glouton_valeur(10, [5, 4, 7], [4, 4, 6]) # retourne 6
```

Est-ce que cet algorithme est toujours optimal ?

7. De même, écrire une **fonction glouton_ratio(c, poids, val)** qui renvoie la valeur totale des objets choisis par l'algorithme glouton, en considérant les objets dans l'ordre de ratio valeur/poids décroissant . On pourra utiliser deux fois combine.
Exemple :

```
glouton_ratio(10, [5, 4, 7], [4, 4, 6]) #8
```

Programmation dynamique

Pour résoudre ce problème de maximisation de la valeur emportée, par programmation dynamique, nous allons utiliser une approche itérative ascendante en stockant les calculs dans une matrice dp de n+1 lignes et c+1 colonnes, n désignant le nombre d'objets disponibles et c la capacité du sac.
Soit dp[i][j] la valeur maximale que l'on peut mettre dans un sac de capacité i, en ne considérant que les j premiers objets. On suppose que les poids sont strictement positifs.

8. Que vaut **dp[i][0]** ?

9. Exprimer **dp[i][j] en fonction de dp[i][j-1]** dans le cas où $poids_j$>i.

10. On suppose que $poids_j$ <=i. Déterminer une **formule de récurrence sur dp[i][j]**, en distinguant le cas où l'objet j est choisi et le cas où il ne l'est pas.

11. En déduire une **fonction prog_dyn(c, poids, val)** qui renvoie la valeur maximale que l'on peut emporter dans un sac de capacité c, en ne considérant que les premiers objets, en remplissant une matrice dp de taille (c+1) x (n+1).

Aide : Evolution de la matrice dp au cours des itérations :
Pour C = 10, poids = [5, 4, 7], valeur = [4, 4, 6]

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 4, 4, 4, 4, 4, 4], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 4, 4, 4, 4, 4, 4], [0, 0, 0, 0, 4, 4, 4, 4, 8, 8], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 4, 4, 4, 4, 4, 4], [0, 0, 0, 0, 4, 4, 4, 4, 8, 8], [0, 0, 0, 0, 4, 4, 4, 6, 6, 8, 8]]
```

Comparaison

12. Écrire une fonction **genere_instance()** qui renvoie un triplet (c, poids, v), où c est un entier aléatoire compris entre 1 et 1000 et poids, v sont des listes de 100 entiers aléatoires entre 1 et 100.
Pour rappel, la fonction randint(nb1, nb2) du module random génère un entier aléatoire entre nb1 et nb2 inclus.

13. Afficher, pour chaque stratégie gloutonne (ordre de poids, ordre de valeur, ordre de ratio**), l'erreur commise par rapport à la solution optimale**, en moyennant sur 100 instances générées par genere_instance().
Quelle stratégie gloutonne est la plus efficace ?

14. Comparer le temps total d'exécution de la stratégie gloutonne par ratio et de la programmation dynamique, sur 100 instances générées par genere_instance(). On pourra importer time et utiliser time.time() pour obtenir le temps actuel en secondes.

Obtention de la liste des objets choisis pour la solution optimale

15. **Réécrire la fonction prog_dyn(c, poids, val)** pour qu'elle renvoie la liste des objets choisis.
Pour cela, on peut construire la matrice dp et remarquer que :

- si dp[i][j] = dp[i][j-1] alors l'objet j n'est pas choisi
- si dp[i][j] = dp[i-poidsj][j-1] +vj alors l'objet j est choisi

On peut donc construire la liste des objets choisis en remontant la matrice dp à partir de la case (c,n)