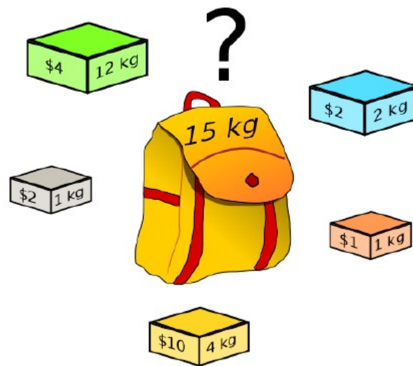
TD Sac à dos

On rappelle le problème du sac à dos :

- Entrée : une capacité c et des objets de poids $w_1, \ldots, w_n$ et valeurs $v_1, \ldots, v_n$.
- Sortie : la valeur maximum que l'on peut mettre dans un sac de capacité c.
  c est le poids total maximum que l'on peut peut mettre dans le sac



L'objectif du TP est de comparer un algorithme de programmation dynamique à différents algorithmes gloutons.

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 dans lequel on choisit les objets, on obtient des algorithmes gloutons différents.

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

Tester avec l'exemple ci-dessous.
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 L1 et L2 ont la même longueur.
   Par exemple,

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

   renvoie :

   ```
   [(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.
   Par exemple

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

   renvoie :

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

   Si L est une liste, ***L.sort()*** trie L par ordre croissant (L.sort (reverse=True) pour 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()
   print(L) # trie suivant le 1er element de chaque couple
   ```

   affiche :

   ```
   [(1, 4), (3, 6), (7, 5)]
   ```

4. Écrire une fonction ***tri_poids(w, v)*** qui renvoie les listes $w2$ et $v2$ obtenues à partir de w et v en triant les poids par ordre croissant.
   On pourra utiliser *L.sort, combine et split.*
   Par exemple :

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

   renvoie :

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

Stratégies gloutonnes

5. Écrire une fonction ***glouton_poids(c, w, v)*** qui renvoie la valeur totale des objets choisis par l'algorithme glouton, en considérant les objets dans l'ordre de poids croissant.
   On pourra réutiliser *glouton*.
   Est-ce que cet algorithme est toujours optimal ?
   Par exemple :

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

   renvoie 8.

6. Écrire de même des fonctions ***tri_valeur (w, v)*** et ***glouton_valeur(c, w, v)*** 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)).
   Est-ce que cet algorithme est toujours optimal ?
   Par exemple

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

   renvoie 6.

7. De même, écrire une fonction ***glouton_ratio(c, w, v)*** 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*.
   Par exemple :

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

   renvoie 8.

Programmation dynamique

On rappelle le problème du sac à dos :

- Entrée : une capacité c et des objets de poids $w_1, \ldots, w_n$ et valeurs $v_1, \ldots, v_n$.
- Sortie : la valeur maximum que l'on peut mettre dans un sac de capacité c.

Soit $dp[i][j]$ la valeur maximum que l'on peut mettre dans un sac de capacité j, en ne considérant que les i premiers objets.
On suppose que les poids sont strictement positifs.

8. Que valent $dp[i][0]$ et $dp[0][j]$?
9. Exprimer $dp[i][j]$ en fonction de $dp[i-1][j]$ dans le cas où $w_i > j$.

10. Supposons $w_i \leq j$. Donner 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. On supposera que la capacité c est un entier naturel.
    Écrire une fonction ***prog_dyn(c, w, v)*** qui renvoie la valeur maximum que l'on peut mettre dans un sac de capacité c, en ne considérant que les i premiers objets, en remplissant une matrice *dp* de taille $(n+1) \times (c+1)$.
    Par exemple,

    ```
    prog_dyn(10, [5, 4, 7], [4, 4, 6])
    ```

    renvoie 8.

Comparaison

12. Écrire une fonction ***genere_instance()*** qui renvoie un triplet (c, w, v), où c est un entier aléatoire entre 1 et 1000 et w, v sont des listes de 100 entiers aléatoires entre 1 et 100.

    On importera *random* pour utiliser *random.randint(a, b)* qui génère un entier aléatoire entre a et b 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.
15. Réécrire la fonction ***prog_dyn (c, w, v)*** pour qu'elle renvoie la liste des objets choisis. Pour cela, on peut construire la matrice *dp* et remarquer que :
    - si $dp[i][j] = dp[i-1][j]$, alors l'objet i n'est pas choisi ;
    - si $dp[i][j] = dp\,[i-1]\,[j-w_i] + v_j,$ alors l'objet i est choisi.
      On peut donc construire la liste des objets choisis en remontant la matrice *dp* à partir de la case (n, c).