TD : PROGRAMMATION DYNAMIQUE
== PROBLÈME DU SAC À DOS ==

Remarque : les rappels théoriques sont en dernière page de ce sujet.

Le fichier source à utiliser pour ce TD est : « TD2 – SacADos.py »

Un randonneur prépare son sac à dos pour une expédition en montagne. Il dispose de plusieurs équipements, chacun ayant une valeur d'utilité (importance pour la randonnée) et un poids. Son sac à dos a une capacité maximale limitée, et il souhaite emporter les équipements qui maximisent la valeur totale d'utilité tout en respectant la contrainte de poids.

Objet	Équipement	Valeur	Poids (kg)
1	Tente légère	7	4
2	Sac de couchage	5	3
3	Réchaud + gamelle	3	2
4	Nourriture (3 jours)	6	5
5	Trousse de secours	4	2

La capacité maximale du sac à dos est C = 10 kg. L'objectif est de trouver le sous-ensemble d'équipements qui maximise la valeur totale tout en respectant cette contrainte de poids.

L'objectif de ce TD est d'implémenter les algorithmes de programmation dynamique (approches bottom-up et top-down) pour résoudre ce problème, puis reconstruire la solution optimale. Vous utiliserez des dictionnaires Python pour mémoriser les résultats des sous-problèmes

I) APPROCHE BOTTOM-UP (TABULATION)

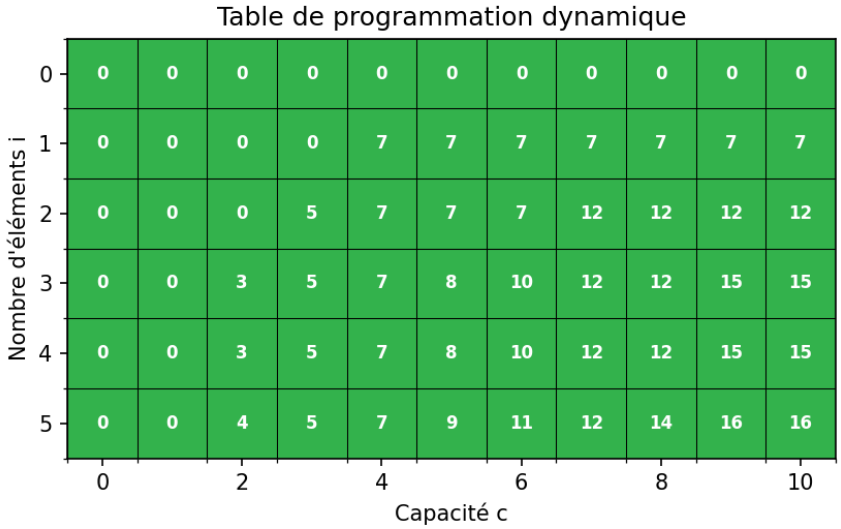
Dans cette partie, vous allez implémenter l'approche bottom-up qui remplit une table de tous les sous-problèmes, des plus petits aux plus grands. Les données sont déjà définies dans le fichier source sous la forme d'un dictionnaire `objets = {n°objet:(Valeur, Poids)}` :

```
objets = {1:(7,4),2:(5,3),3:(3,2),4:(6,5),5:(4,2)}
C = 10    # Capacité maximale
A = {}    # Table de mémoïsation
```

1. Écrire une fonction `initialiser_table(A, objets, C)` qui initialise le dictionnaire représentant la table A. La fonction doit uniquement initialiser les cas de base :
   - A[(0, c)] = 0 pour tout c de 0 à C (aucun objet disponible)
   - A[(i, 0)] = 0 pour tout i de 0 à n (capacité nulle)

2. Écrire une fonction `remplir_table(A, objets, C)` qui remplit entièrement la table A en utilisant l'équation de récurrence du cours et qui est rappelée à la fin du sujet. Attention à l'ordre de parcours : on doit calculer A[(i, c)] pour i allant de 1 à n, et pour chaque i, c allant de 1 à C.

Vérifier : `>>> AfficheTable(A,objets,C)`



5. Combien de sous-problèmes sont calculés dans l'approche bottom-up ? Quelle est la complexité temporelle de cet algorithme ?

II) APPROCHE TOP-DOWN AVEC MÉMOÏSATION

Dans cette partie, vous allez implémenter l'algorithme récursif avec mémoïsation. L'idée est de partir du problème principal et de le décomposer en sous-problèmes, en mémorisant les résultats pour éviter les calculs redondants.

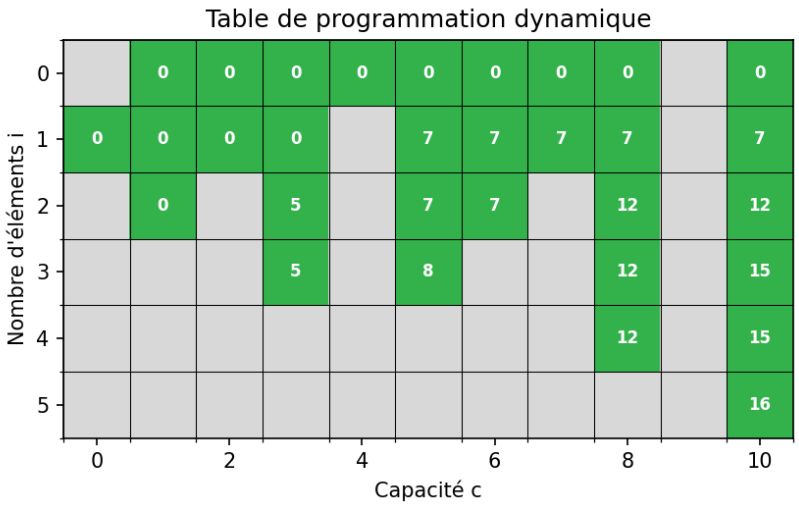
On utilisera un dictionnaire global pour la mémoïsation : `A = {}`

1. Écrire une fonction récursive `rec_opt_val(i, c)` qui implémente la récurrence avec mémoïsation rappelé à la fin du sujet.

Tester :

```
>>> rec_opt_val(0,0)
0
>>> rec_opt_val(3,3)
5
>>> rec_opt_val(0,10)
0
>>> rec_opt_val(3,5)
8
```

2. Initialiser le dictionnaire de la table puis appeler la fonction précédente afin de chercher la valeur optimale. Vérifier votre table :



3. Comparez les tables obtenues par les approches bottom-up et top-down. Que constatez-vous concernant le nombre de cases remplies ? Expliquez cette différence.

4. Quelle est la complexité temporelle de l'algorithme top-down avec mémoïsation ? Justifiez votre réponse en considérant le nombre de sous-problèmes distincts et le coût de chaque sous-problème.

5. Quelle est la complexité spatiale de cet algorithme ? Prenez en compte à la fois le dictionnaire de mémoïsation et la pile d'appels récursifs.

III) RECONSTRUCTION DE LA SOLUTION

Maintenant que nous connaissons la valeur optimale, nous devons déterminer quels équipements le randonneur doit emporter. Cette étape s'appelle la reconstruction de la solution.

La reconstruction consiste à « remonter » dans la table A pour déterminer, à chaque étape, si l'objet i a été pris ou non. On part de A[(n, C)] et on remonte jusqu'à i = 0.

1. Écrire une fonction `objet_pris(A, objets, i, c)` qui retourne `True` si l'objet i a été pris pour obtenir la valeur `A[(i, c)]`, `False` sinon.

```
Tester :    >>> objet_pris(A,5,10)            >>> objet_pris(A,4,8)
            True                              False
```

2. Écrire une fonction `reconstruire_solution(A, objets, C)` qui retourne la liste des indices des objets faisant partie de la solution optimale.

```
Vérifier :  >>> reconstruire_solution(A,objets,C)
            [5, 2, 1]
```

3. Quelle est la complexité temporelle de l'algorithme de reconstruction ?

4. Vérifiez que la solution obtenue respecte bien la contrainte de capacité. Que remarquez-vous concernant le poids total par rapport à la capacité maximale ?

RAPPELS THÉORIQUES

Formulation du problème

Une instance du problème du sac à dos est spécifiée par (2n + 1) entiers positifs, où n est le nombre d'objets : une valeur v_i et une taille s_i pour chaque objet i, ainsi qu'une capacité C du sac à dos.

La tâche de l'algorithme est de sélectionner un sous-ensemble S d'objets tel que la valeur totale $\sum_{i \in S} v_i$ soit maximale, sous la contrainte que la taille totale $\sum_{i \in S} s_i$ soit au plus C.

Sous-problèmes et notation

On définit le sous-problème $A_{i,c}$ comme suit :

- $A_{i,c}$ = valeur totale maximale d'un sous-ensemble des i premiers objets dont la taille totale est au plus c.
- Quand i = 0, on interprète $A_{0,c}$ comme étant 0.

Relation de récurrence

Pour calculer $A_{i,c}$, on distingue deux cas selon que l'on prend ou non l'objet i :

- Cas n°1 (on ne prend pas l'objet i) : $A_{i,c} = A_{i-1,c}$
- Cas n°2 (on prend l'objet i) : $A_{i,c} = A_{i-1,c-si} + v_i$

La récurrence s'écrit donc :

$$A_{i,c} = \begin{cases} A_{i-1,c} & s_i > c \\ max \begin{cases} A_{i-1,c} \\ A_{i-1,c-s_i} + v_i \end{cases} & s_i \leq c \end{cases}$$

Cas de base

Les cas de base sont les suivants :

- $A_{0,c} = 0$ pour tout c : aucun objet disponible, valeur nulle.
- $A_{i,0} = 0$ pour tout i : capacité nulle, on ne peut prendre aucun objet.

Algorithme de reconstruction

Une fois la table A remplie, on reconstruit la solution optimale en « remontant » dans la table depuis $A_{n,C}$ jusqu'à i = 0.

Principe : Pour chaque objet i (de n à 1), on détermine s'il a été pris ou non :

- Si $A_{i,c} \neq A_{i-1,c}$, l'objet i a été pris. On l'ajoute à la solution et on met à jour c := c - s_i.
- Sinon, l'objet i n'a pas été pris. On passe à l'objet suivant sans modifier c.