Programmation dynamique – Activités d'introduction

Activité 1 : la suite de Fibonacci

La célèbre suite de Fibonacci est définie par $F_0 = 0$, $F_1 = 1$ et pour tout entier naturel $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$.

Pour les non matheux, il s'agit de la suite de nombres 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 …
Chaque nombre s'obtient en ajoutant les deux précédents. Le premier terme de cette liste est dit terme de rang ou d'indice 0, le suivant terme de rang 1, … Ainsi, le terme de rang 7 est égal à 13. On le note F_7.

Écrire une fonction récursive fibo(n) renvoyant le nombre d'indice *n* de la suite de Fibonacci, le premier de la liste ayant pour indice 0.

Utiliser votre fonction pour calculer les termes d'indice 10, 20, 30, 40… quel est le problème ?

Activité 2 : le sac à dos

On rappelle le problème du sac à dos déjà vu en première : on dispose de *n* objets de valeurs et de poids données, assimilables à des couples (v_0, p_0), (v_1, p_1), ..., (v_{n-1}, p_{n-1}), et d'un sac à dos qui peut porter un poids maximum *w*. L'objectif est de maximiser la valeur des objets contenus dans le sac.

Vous avez vu deux stratégies en première :

- force brute : tester toutes les combinaisons possibles, envisageable avec 20 objets par exemple, mais pas avec 60 objets.
- algorithmes gloutons :
    - glouton 1 : on prend d'abord les objets de valeurs maximales
    - glouton 2 : on prend d'abord les objets maximisant le rapport valeur/poids.

Les algorithmes gloutons sont très rapides, en $O(n \log_2(n))$ si on trie les objets suivant le critère choisi avec un bon algorithme de tri, mais ne garantissent pas d'obtenir la meilleure solution.

Approche top-down – Solution récursive

On peut construire une solution optimale du problème à *i* objets à partir d'une résolution du problème à $i - 1$ objets, ce qui fournit une solution récursive au problème.
En effet, pour obtenir une solution optimale du problème à *i* objets avec le poids maximal *p* :

- ou bien on ne prend pas le *i*-ième objet, et on cherche une solution optimale du problème à $i - 1$ objets avec le poids maximal *p*,
- ou bien on prend le *i*-ième objet, de poids p_i et on cherche pour compléter une solution optimale du problème à $i - 1$ objets avec le poids maximal $p - p_i$.

TP Q1. Ouvrir le fichier sacados.py. Compléter la fonction sac_a_dos_rec.
Lancer la fonction test_rec() pour tester votre code.

Le problème de la solution récursive est qu'elle effectue plusieurs fois les mêmes calculs. Pour l'éviter, on va stocker les résultats des appels déjà effectués. On peut utiliser un dictionnaire ou un tableau.

TP Q2. Écrire la foncion sac_a_dos_mem_dico à partir de la fonction sac_a_dos_rec en stockant les résultats des calculs intermédiaires dans un dictionnaire. Tester votre code avec la fonction test_mem_dico().

TP Q3. Écrire la fonction sac_a_dos_mem_tab en remplaçant le dictionnaire par un tableau à deux dimensions. Tester votre code avec la fonction test_mem_tab().

Il est inutile de stocker les listes d'objets à chaque étape du calcul. Il suffit de stocker les valeurs maximales. En effet, on peut ensuite reconstituer la liste d'objets à partir du tableau des valeurs maximales. On obtient un code moins gourmand en mémoire et plus rapide.

TP Q4. Écrire la fonction sac_a_dos_mem_val qui ne renvoie et ne mémoïse dans le tableau que les valeurs maximales. Tester votre code avec la fonction test_mem_val().

La version définitive de cette approche top-down appelle la fonction sac_a_dos_mem_val pour construire le tableau avec les valeurs maximales. Il faut ensuite reconstituer la liste d'objets.
Pour cela, on démarre la lecture du tableau en [n][w].
Pour savoir s'il faut prendre le n-ième objet (d'indice $n - 1$), on compare tab[n][w] avec tab[n-1][w].
Si les deux nombres sont égaux, on obtient la valeur maximale sans prendre le n-ième objet et on poursuit la lecture en [n-1][w] pour s'intéresser à l'objet précédent.
Si les deux nombres sont différents, cela signifie que la valeur maximale s'obtient en prenant le dernier objet de poids p. On ajoute donc cet objet à la liste d'objets sélectionnés et on poursuit la lecture du tableau en [n-1][w-p] pour s'intéresser à l'objet précédent.
On poursuit ce procédé jusqu'au premier objet.

TP Q5. Écrire la fonction sac_a_dos_mem_final qui utilise comme expliqué ci-dessus la fonction sac_a_dos_mem_val et renvoie la valeur maximale et la liste d'objets associés. Tester votre code avec la fonction test_mem_final().

Approche bottom-up

On résout successivement les problèmes à 1 objet, 2 objets, 3 objets, … pour les poids allant de 0 à w. On présente les solutions (valeurs maximales) dans un tableau. Le contenu du tableau dépend de l'ordre des objets mais pas la dernière ligne.
La première ligne, correspondant à 0 objet, est remplie trivialement.
Ensuite, on remplit une ligne à partir de la ligne précédente en utilisant le même principe que dans l'approche récursive, que l'on peut écrire sous forme de relation de récurrence. En notant $t_{i,j}$ la valeur maximale possible avec i objets et un poids maximal j, on a la relation :

$t_{i,j} =$

Exemple
Résolution du problème du sac à dos avec la liste objets = [(3, 2), (8, 10), (2, 2), (8, 1), (4, 6), (6, 6)] et le poids maximal w = 10 kg. Les objets sont au format (valeur, poids).

objets\poids	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3	3
2	0	0	3	3	3	3	3	3	3	3	8
3	0	0	3	3	5	5	5	5	5	5	8
4	0	8	8	11	11	13	13	13	13	13	13
5	0	8	8	11	11	13	13	13	13	15	15
6	0	8	8	11	11	13	13	14	14	17	17

La valeur maximale est 17, atteinte avec un poids de 9 kg. Puisque cette valeur n'est pas atteinte avec 5 objets, on a pris l'objet n°6, qui pèse 6 kg, donc il reste 9 – 6 = 3 kg pour 5 objets. Pour 5 objets, la valeur maximale atteinte avec 3 kg est égale à 11, c'est la même avec 4 objets. On n'a donc pas pris l'objet n°5, mais on a pris l'objet n°4 qui pèse 1 kg, donc il reste 2 kg pour 3 objets, ce qui permet une valeur égale à 3, déjà atteinte avec l'objet n°1.

On obtient donc la valeur optimale de 17 avec les objets 1, 4, 6.

Exercice
Même exercice avec objets = [(5, 3), (9, 2), (10, 5), (6, 4), (7, 1), (9, 3)] et $w = 10$.

Algorithme

1. Écrire l'algorithme en langage naturel permettant, à partir d'une liste d'objets au format (valeur, poids) et d'un poids maximal w de construire le tableau des solutions du problème du sac à dos comme ci-dessus.

2. Écrire l'algorithme renvoyant une solution optimale à partir du tableau précédent.

3. Quelle est la complexité, en temps et en mémoire, de cette méthode de résolution ?

Programmation

TP Q6. Écrire la fonction sac_a_dos_dyn_tab qui renvoie le tableau des solutions des problèmes de 0 à n objets pour des poids allant de 0 à w en utilisant la programmation dynamique bottom-up. Tester votre code avec la fonction test_dyn_tab().

TP Q7. Écrire la fonction sac_a_dos_dyn qui utilise la fonction sac_a_dos_dyn_tab et renvoie la valeur maximale et une liste d'objets réalisant cette valeur. Tester votre code avec la fonction test_dyn().