COURS : PROGRAMMATION DYNAMIQUE
= LE PROBLÈME DU SAC À DOS =

Notre troisième étude de cas concerne le problème du sac à dos. Nous allons construire la solution du problème par programmation dynamique.

I) DÉFINITION DU PROBLÈME 2
II) SOUS-STRUCTURE OPTIMALE ET RELATION DE RÉCURRENCE 3
II.1. Sous-structure optimale 3
II.2. Équation de récurrence sur les valeurs optimales 3
III) SOUS-PROBLÈMES ET COMPLEXITÉ 4
III.1. Définition des sous-problèmes 4
III.2. Schéma de récursion 5
III.3. Complexité sans mémoïsation 5
IV) ALGORITHMES DE PROGRAMMATION DYNAMIQUE 6
IV.1. Algorithme top-down 6
IV.2. Complexité de l'algorithme top-down 7
IV.3. Algorithme bottom-up 7
IV.4. Complexité de l'algorithme bottom-up 8
V) ALGORITHME DE RECONSTRUCTION 8
V.1. Principe et algorithme de reconstruction 8
V.2. Complexité finale 9

I) DÉFINITION DU PROBLÈME

Une instance du problème du sac à dos est spécifiée par (2n + 1) entiers positifs, où n est le nombre d'objets (qui sont étiquetés arbitrairement de 1 à n) : 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 d'objets. La valeur totale des objets doit être aussi grande que possible tout en tenant dans le sac à dos, ce qui signifie que leur taille totale doit être au plus égale à C.

Problème du sac à dos

Entrée : les valeurs des objets v_1, v_2, …, v_n, les tailles des objets s_1, s_2, …, s_n, et une capacité C du sac à dos (tous des entiers positifs).

Sortie : un sous-ensemble $S \subseteq \{1, 2, \ldots, n\}$ d'objets tel que la somme des valeurs $\sum_{i \in S} v_i$ soit maximale, sous la contrainte que la taille totale $\sum_{i \in S} s_i$ soit au plus C.

Exemple : soit un problème du sac à dos avec une capacité de sac C = 6 et quatre objets :

Objet	Valeur	Taille
1	3	4
2	2	3
3	4	2
4	4	3

Comme la capacité du sac à dos est de 6, il n'y a pas de place pour choisir plus de deux objets. La paire d'objets la plus intéressante (la plus grande valeur) est constituée du troisième et du quatrième (avec une valeur totale de 8), et ceux-ci tiennent dans le sac à dos (avec une taille totale de 5).

Chaque fois qu'on a une ressource rare que l'on veut utiliser de la manière la plus intelligente possible, c'est un problème de sac à dos. Par exemple, sur quels biens et services dépenser son salaire pour en tirer le maximum de valeur ? Ou encore, étant donné un budget de fonctionnement et un ensemble de candidats à l'embauche avec des productivités différentes et des salaires demandés variés, lesquels recruter ?

Pour résoudre ce problème de manière exhaustive (par brute force), il faut :

- Lister tous les sous-ensembles possibles d'objets ;
- Pour chacun, calculer la taille totale, vérifier si cela tient dans le sac, puis regarder la valeur totale ;
- Garder le meilleur.

Pour n objets, il y a 2^n sous-ensembles possibles (chaque objet est soit pris, soit non pris). Pour chaque sous-ensemble, calculer les tailles et les valeurs coûte O(n). Au total, la complexité du problème est donc de $O(n \cdot 2^n)$. C'est un problème exponentiel en n qui demande un espace mémoire en O(n) pour stocker le sous-ensemble courant et le meilleur trouvé.

II) SOUS-STRUCTURE OPTIMALE ET RELATION DE RÉCURRENCE

Pour appliquer le principe de la programmation dynamique au problème du sac à dos, nous devons déterminer la bonne collection de sous-problèmes. Nous y parviendrons en raisonnant sur la structure des solutions optimales et en identifiant les différentes façons dont elles peuvent être construites à partir de solutions optimales de sous-problèmes plus petits.
Nous pourrons ensuite établir une relation de récurrence permettant de calculer rapidement la solution d'un sous-problème à partir de celles de deux sous-problèmes plus petits.

II.1. Sous-structure optimale

Considérons une instance du problème du sac à dos avec les valeurs des objets v_1, v_2, …, v_n, les tailles des objets s_1, s_2, …, s_n et une capacité de sac C. Supposons que quelqu'un nous donne, sur un plateau, une solution optimale $S \subseteq \{1, 2, \ldots, n\}$ de valeur totale $V = \sum_{i \in S} v_i$. À quoi doit-elle ressembler ?

Commençons par nous demander : soit S contient le dernier objet (l'objet n), soit elle ne le contient pas :

Cas n°1 : $n \notin S$: supposons que la solution optimale S ne contient pas le dernier objet n.

Comme la solution optimale S exclut le dernier objet, on peut la considérer comme une solution réalisable (toujours avec une valeur totale V et une taille totale au plus égale à C) du problème plus petit ne comportant que les (n – 1) premiers objets (et une capacité de sac C).

Cas n°2 : $n \in S$: supposons que la solution optimale S contient le dernier objet n.

Ce cas ne peut se produire que lorsque $s_n \leq C$. Nous ne pouvons pas considérer S comme une solution réalisable à un problème plus petit ne comportant que les (n – 1) premiers objets, mais nous le pouvons après avoir retiré l'objet n. Dans ce cas, S – {n} est une solution optimale à un sous-problème plus petit, constitué des (n – 1) premiers objets et d'une capacité de sac $C – s_n$, avec une valeur totale $V – v_n$.

II.2. Équation de récurrence sur les valeurs optimales

Nous avons vu que si S est une solution optimale d'un problème de sac à dos avec $n \geq 1$ objets, de valeurs v_1, v_2, …, v_n, de tailles s_1, s_2, …, s_n et de capacité de sac C, alors S est soit :

- une solution optimale pour les (n – 1) premiers objets avec une capacité de sac C,
- une solution optimale pour les (n – 1) premiers objets avec une capacité de sac $C – s_n$, complétée par le dernier objet n.

La première solution est toujours une possibilité pour la solution optimale. La seconde solution est une possibilité si et seulement si $s_n \leq C$. Dans ce cas, s_n unités de capacité sont effectivement réservées à l'avance pour l'objet n. L'option ayant la valeur totale la plus grande est une solution optimale.

Cela conduit à la relation de récurrence suivante :

Récurrence sur la valeur de la solution optimale

Notons $V_{i,c}$ la valeur totale maximale d'un sous-ensemble des i premiers objets dont la taille totale est au plus c. (Quand i = 0, on interprète $V_{i,c}$ comme étant 0).
Pour tout i = 1, 2, …, n et tout c = 0, 1, 2, …, C :

$$V_{i,c} = \begin{cases} V_{i-1,c} & s_i > c \\ max\left\{ \underbrace{V_{i-1,c}}_{cas\ n°1}, \underbrace{V_{i-1,c-s_i}}_{cas\ n°2} + v_i \right\} & s_i \leq c \end{cases}$$

III) SOUS-PROBLÈMES ET COMPLEXITÉ

III.1. Définition des sous-problèmes

L'étape suivante consiste à définir la collection de sous-problèmes pertinents et à les résoudre systématiquement en utilisant la relation de récurrence. Pour l'instant, nous nous concentrons sur le calcul de la valeur totale d'une solution optimale pour chaque sous-problème. Nous pourrons reconstruire les objets d'une solution optimale du problème original à partir de ces informations.

Pour le problème du sac à dos, nous voyons que les sous-problèmes doivent être paramétrés par deux indices : la longueur i du préfixe des objets disponibles et la capacité c disponible du sac à dos.

En faisant varier ces deux paramètres sur toutes les valeurs pertinentes, nous obtenons nos sous-problèmes :

Sous-problèmes du sac à dos

Calculer $V_{i,c}$, la valeur totale d'une solution optimale au problème du sac à dos utilisant les i premiers objets et une capacité de sac c.

(Pour chaque i = 0, 1, 2, …, n et c = 0, 1, 2, …, C)

Le plus grand sous-problème (avec i = n et c = C) est exactement le même que le problème original. Comme toutes les tailles d'objets et la capacité C du sac à dos sont des entiers positifs, et comme la capacité est toujours réduite de la taille d'un objet (pour lui réserver de la place), les seules capacités résiduelles qui peuvent apparaître sont les entiers compris entre 0 et C.

III.2. Schéma de récursion

Le schéma de récursion sur un exemple avec 3 objets est donné sur la figure ci-dessous :

- La notation [1,2,3][6] signifie qu'on cherche la solution optimale au problème constitué des trois premiers objets [1,2,3], la capacité restante étant de [6].
- La notation A[3][6]=7 signifie que la valeur optimale du sous-problème contenant les [3] premiers objets avec une capacité de [6] vaut 7.
- Dans les cas n°1 (branches de gauche), à partir des cas de base du bas (en vert) on remonte la valeur A[i-1][c] ;
- Dans les cas n°2 (branches de droite), on remonte la valeur A[i-1][c-s_i] + v_i.
- Les valeurs optimales A[·][·] prennent le maximum entre les deux valeurs remontées.

https://www.informatique-f1.fr/dp/sacados/

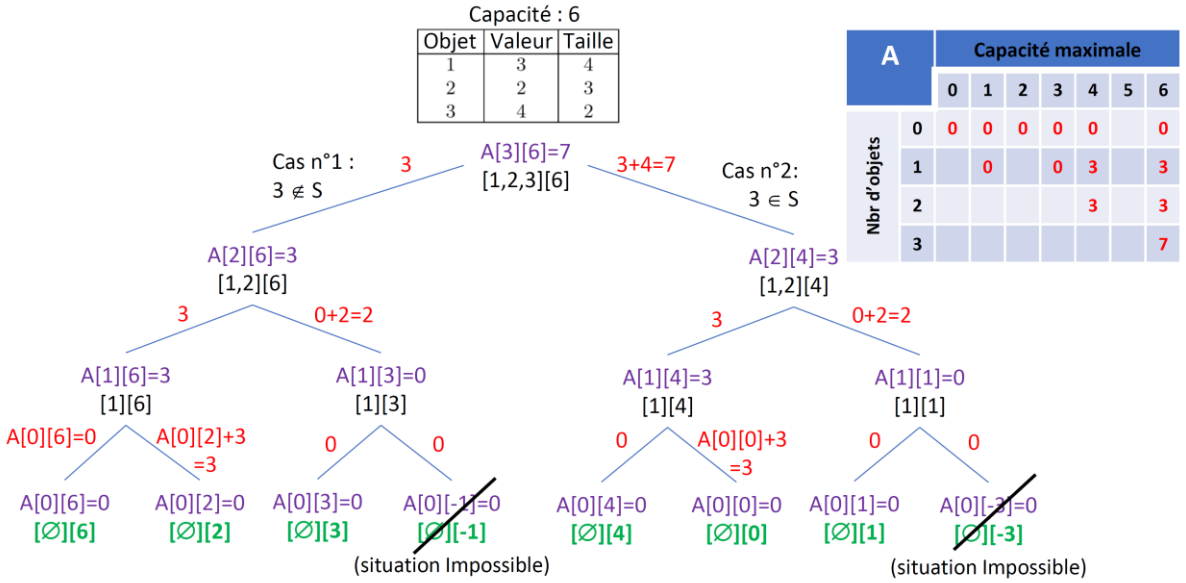

Figure 1 : Schéma de récursion du problème du sac à dos

III.3. Complexité sans mémoïsation

Chaque niveau de récursivité ne peut enlever qu'un seul objet. Il faut donc descendre jusqu'au niveau n pour avoir des cas de base. Tous les nœuds jusqu'au niveau (n – 1) sont donc des nœuds internes qui se ramifient encore, avec un facteur de branchement égal à 2 (si on ne tient pas compte des cas où $s_i > c$). Le nombre exact de nœuds dépend de l'instance du problème. Dans le cas le plus défavorable, il peut aller jusqu'à 2^n au niveau n. Le nombre total de nœuds est alors de $1 + 2 + 4 + \ldots + 2^n = 2^{n+1} – 1$.

À chaque nœud de l'arbre de récursion, le travail local (hors appels récursifs) se fait en temps O(1) : on effectue seulement un nombre constant d'opérations (comparaisons, additions, calcul d'un maximum). Comme l'arbre de récursion est binaire et peut contenir jusqu'à $O(2^n)$ nœuds dans le pire des cas, le temps d'exécution de cet algorithme récursif sans mémoïsation est exponentiel, en $O(2^n)$.

Remarquons qu'en pratique, l'algorithme top-down ne résout que les sous-problèmes qui sont réellement atteints en partant de l'état initial (n, C) et en suivant la récurrence. Certains couples (i, c) ne sont jamais visités : par exemple parce que certaines capacités ne peuvent pas apparaître (combinaisons impossibles), ou parce que des branches sont coupées quand un objet ne rentre pas ($s_i > c$).

IV) ALGORITHMES DE PROGRAMMATION DYNAMIQUE

IV.1. Algorithme top-down

Étant donnés les sous-problèmes et la relation de récurrence, on peut mettre en place un algorithme top-down (avec mémoïsation) de programmation dynamique pour le problème du sac à dos.

Algorithme top-down pour le calcul des valeurs optimales

Entrée : v[1, …, n] : valeurs des objets
s[1, …, n] : tailles des objets
C : capacité maximale du sac à dos

Sortie : valeur totale maximale d'un sous-ensemble S ⊆ {1, 2, …, n} tel que $\sum_{i \in S} s_i \leq C$.

```
# Valeurs maximales des sous-ensembles stockées dans un dictionnaire
A := {}

rec_opt_val_SAC (i, c) :
    # i : nombre d'objets considérés (les i premiers)
    # c : capacité restante

    # Utilise la mémoïsation
    Si (i, c) est dans A :
        Retourner A[(i, c)]

    # Cas de base quand i = 0 ou c = 0: A[(0, c)] = 0 ; A[(i, 0)] = 0
    Si i == 0 ou c == 0 :
        A[(i, c)] := 0
        Retourner A[(i, c)]

    # Récursion cas n°1 : on ne prend pas l'objet i
    S1 := rec_opt_val_SAC (i - 1, c)          # Cas n°1 : V_{i,c} = V_{i-1,c}

    # Si s[i] > c alors retourne la solution S1
    Si s[i] > c :
        A[(i, c)] := S1
        Retourner A[(i, c)]

    # Cas n°2 : on prend l'objet i
    S2 := rec_opt_val_SAC (i - 1, c - s[i]) + v[i]    # Cas n°2 : V_{i,c} = V_{i-1,c-si} + v_i

    # Sauvegarde et retourne la solution optimale
    A[(i, c)] := max(S1, S2)
    Retourner A[(i, c)]

# Appel initial :
résultat := rec_opt_val_SAC (n, C)
```

IV.2. Complexité de l'algorithme top-down

Chaque sous-problème est défini par deux paramètres : le nombre d'objets considérés (0 à n) et la capacité restante (0 à C). Les états possibles sont donc les couples (i, c) avec $0 \leq i \leq n$ et $0 \leq c \leq C$. Le nombre maximal de sous-problèmes distincts est donc de $(n+1)\cdot(C+1) = O(n\cdot C)$.

Avec la mémoïsation, chaque couple (i, c) est calculé au plus une fois et les appels suivants sur les mêmes couples font uniquement un accès dans le dictionnaire des valeurs en O(1).

Lors de la résolution d'un sous-problème (i, c) non mémorisé, l'algorithme effectue un travail local en O(1) (comparaisons, calcul d'un maximum, addition), ainsi qu'au plus deux appels récursifs vers des sous-problèmes comme (i−1, c) et (i−1, $c-s_i$).

Comme chaque sous-problème est résolu au plus une fois, le nombre total d'appels « réels » est en $O(n\cdot C)$, et la complexité en temps est donc $O(n\cdot C)$.

L'espace mémoire utilisé par le dictionnaire de mémoïsation est en $O(n\cdot C)$ et la profondeur de la pile d'appels récursifs est au maximum de n, soit O(n). Le total de l'espace mémoire est donc dominé par le dictionnaire et est de $O(n\cdot C)$.

IV.3. Algorithme bottom-up

L'algorithme bottom-up consiste à remplir progressivement la table des solutions des sous-problèmes en utilisant la relation de récurrence, en partant des cas de base.

Algorithme bottom-up pour le calcul des valeurs optimales

Entrée : v[1, …, n] : valeurs des objets ; s[1, …, n] : tailles des objets ; C : capacité
Sortie : valeur totale maximale d'un sous-ensemble $S \subseteq \{1, 2, \ldots, n\}$ tel que $\sum_{i\in S} s_i \leq C$.

```
A := {}        # Valeurs maximales des sous-ensembles stockées dans un dictionnaire
opt_val_SAC (v, s, C) :
    # Cas de base
    Pour c allant de 0 à C :
        A[(0, c)] := 0

    # Résout l'ensemble des sous-problèmes
    Pour i allant de 1 à n :
        Pour c allant de 0 à C :
            # Utilise l'équation de récurrence
            Si s[i] > c :
                A[(i, c)] := A[(i-1, c)]
            Sinon :
                A[(i, c)] := max {A[(i-1, c)], A[(i-1, c-s[i])] + v[i]}

    # Retourne la solution optimale sur le problème général
    Retourner A[(n, C)]
```

Les tables construites par les algorithmes top-down et bottom-up sont données ci-dessous :

Capacité : 6

Objet	Valeur	Taille
1	3	4
2	2	3
3	4	2

Top-down

A	Capacité maximale						
	0	1	2	3	4	5	6
Nbr d' objets 0	0	0	0	0	0		0
1		0		0	3		3
2					3		3
3							7

Bottom-up

A	Capacité maximale						
	0	1	2	3	4	5	6
Nbr d' objets 0	0	0	0	0	0	0	0
1	0	0	0	0	3	3	3
2	0	0	0	2	3	3	3
3	0	0	4	4	4	6	7

Figure 2 : Tables des valeurs optimales top-down (gauche) et bottom-up (droite)

IV.4. Complexité de l'algorithme bottom-up

À l'inverse de l'algorithme top-down, l'algorithme bottom-up parcourt systématiquement toute la table A[i, c] pour i = 0..n et c = 0..C, même pour des états qui ne seront jamais « utiles » pour la solution finale. Il effectue donc toujours exactement (n + 1)·(C + 1) calculs, indépendamment de la structure de l'instance.

On a donc la même complexité asymptotique O(n·C) pour les deux approches, mais en pratique le top-down peut faire moins de travail effectif sur certaines instances, alors que le bottom-up remplit la table de manière uniforme, quitte à calculer des sous-problèmes inutiles.

V) ALGORITHME DE RECONSTRUCTION

V.1. Principe et algorithme de reconstruction

On peut reconstruire une solution optimale en retraçant le chemin dans le tableau A une fois rempli.

En partant du plus grand sous-problème, dans le coin inférieur droit, l'algorithme de reconstruction vérifie quel cas de la récurrence a été utilisé pour calculer A[n][C] :

- Si A[n][C] == A[n-1][C], alors c'est le cas n°1 : on ne prend pas l'objet n et on reprend la reconstruction à partir de l'entrée A[n-1][C] ;
- Sinon, c'est le cas n°2 : on prend l'objet n et on reprend la reconstruction à partir de l'entrée A[n-1][C-s_n]

L'algorithme de reconstruction est le suivant :

Algorithme de reconstruction

Entrée : v[1, …, n] : valeurs des objets
s[1, …, n] : tailles des objets
C : capacité maximale du sac à dos
A : dictionnaire des valeurs optimales sous la forme A[(i, c)]
Sortie : Solution optimale du problème (numéros des objets)

Reconstruction (v, s, C, A) :
S := ∅ # Objets de la solution optimale
c := C # Capacité restante

Pour i allant de n à 1 :
Si A[(i, c)] ≠ A[(i-1, c)]:
S := S ∪ {i} # Cas n°2, on inclut l'objet i
c := c – s[i] # Réserve l'espace pour cet objet
Retourner S

La figure ci-dessous illustre ce principe de reconstruction :

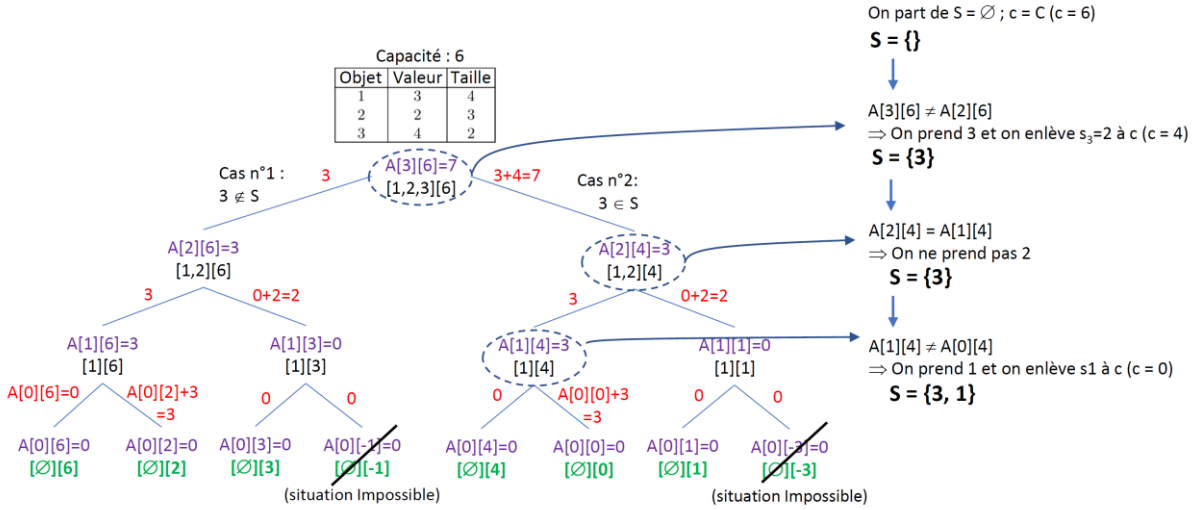

Figure 3 : Principe de reconstruction de la solution optimale

V.2. Complexité finale

L'étape de reconstruction s'exécute en temps O(n) (avec un travail en O(1) par itération de la boucle principale), ce qui est beaucoup plus rapide que le temps O(n·C) nécessaire pour remplir le tableau des valeurs optimales.

Le problème du sac à dos peut donc être résolu par programmation dynamique en temps O(n·C).