PROGRAMMATION DYNAMIQUE

À la fin de ce chapitre, je sais :

- ☞ énoncé les principes de la programmation dynamique
- ☞ distinguer cette méthode des approches gloutonnes et diviser pour régner
- ☞ formuler récursivement le problème du sac à dos

A Motivations

Dans la famille des algorithmes de décomposition, c'est-à-dire les algorithmes qui cherchent à décomposer un problème en sous-problèmes afin de le résoudre, on distingue trois grandes familles :

1. les algorithmes gloutons (cf. chapitre **??**),
2. les algorithmes de type diviser pour régner (cf. chapitre **??**)
3. la programmation dynamique.

La figure 1 schématise ces trois approches sous la forme d'arbres de décomposition de problèmes en sous-problèmes. Les algorithmes de type gloutons ou de type diviser pour régner ont des limites :

1. même s'il existe des algorithmes gloutons optimaux[1], c'est-à-dire qui produisent une solution optimale au problème, la plupart du temps ce n'est pas le cas.
2. même si l'approche diviser pour régner est très efficace pour de nombreux problèmes[2], elle nécessite que les sous-problèmes soient indépendants. Or, parfois, il n'en est rien, certains sous-problèmes ont des sous-problèmes en commun, ils ne sont pas indépendant, ils se chevauchent. Dans ce cas, l'approche diviser pour régner devient inefficace puisqu'elle résout plusieurs fois les mêmes sous-problèmes.

Afin de dépasser ces limites, on étudie la programmation dynamique.

1. On peut citer notamment : la planification de tâches dans le temps qui ne se chevauchent pas, l'algorithme de Prim ou de Huffmann.
2. On peut citer notamment : la transformée de Fourier rapide (FFT), l'exponentiation rapide, les approches dichotomiques.

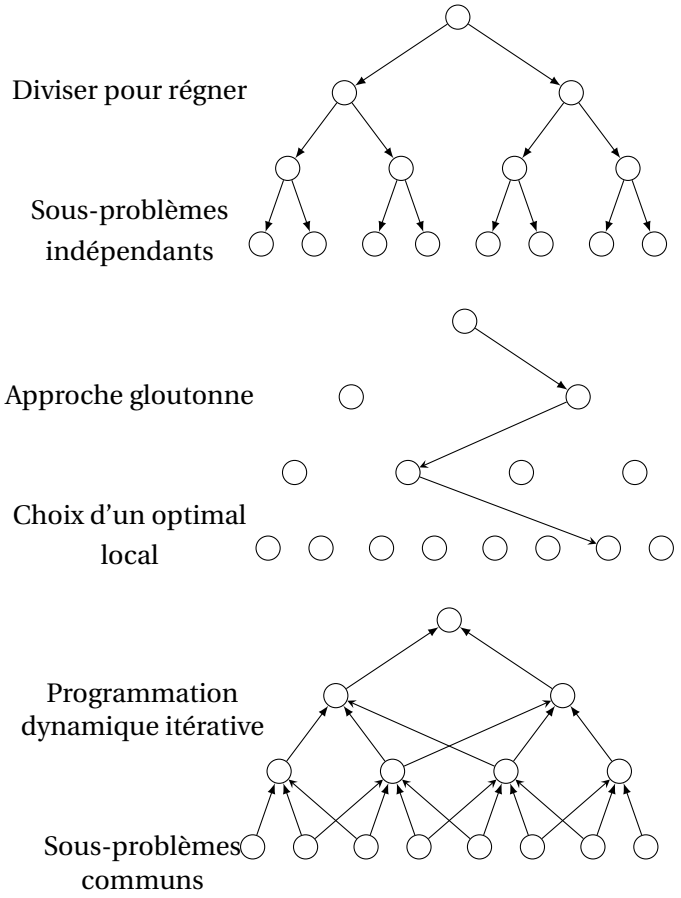

FIGURE 1 – Schématisation des différentes approches des algorithmes de décomposition d'un problème en sous-problèmes : diviser pour régner, approche gloutonne et programmation dynamique itérative.

B Exemples simples de chevauchements des sous-problèmes

■ **Exemple 1 — Calcul de $\binom{n}{k}$.** La formule de récurrence

$$\binom{n}{k} = \begin{cases} 0 & \text{si } k > n \\ 1 & \text{si } k = n \text{ ou } k = 0 \\ n & \text{si } k = 1 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sinon} \end{cases} \tag{1}$$

permet de construire le triangle de Pascal (cf. tableau 1). Cette construction illustre la méthode de complétion d'un tableau de résolution dans le cadre de la programmation dynamique. La figure 2 met en exergue les sous-problèmes et leur imbrication pour calculer $\binom{6}{3}$.

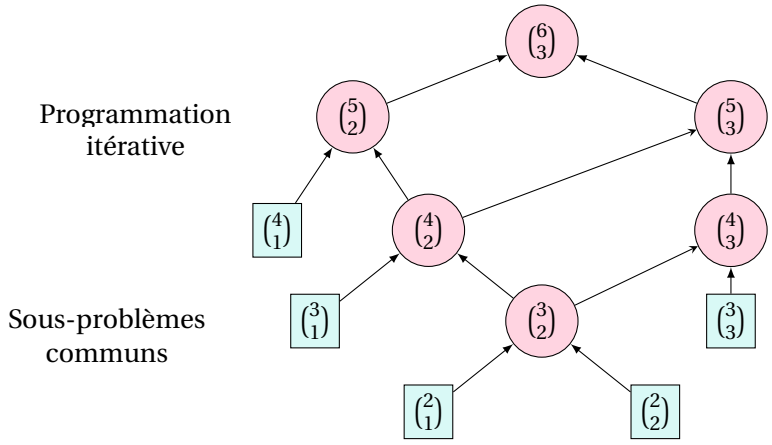

FIGURE 2 – Programmation itérative du calcul de $\binom{6}{3}$. Les rectangles correspondent à un cas terminal de la récursivité, les cercles à l'application de la formule récursive. **Les sous-problèmes se chevauchent** : par exemple, le calcul de $\binom{3}{2}$ est utilisé par le calcul de $\binom{4}{2}$ et $\binom{4}{3}$. En complétant le tableau de résolution, l'idée est de ne le calculer qu'une seule fois ces éléments.

n \ k	0	1	2	3	4	5	6
6	1	6	15	20	15	6	1
5	1	5	10	10	5	1	0
4	1	4	6	4	1	0	0
3	1	3	3	1	0	0	0
2	1	2	1	0	0	0	0
1	1	1	0	0	0	0	0
0	1	0	0	0	0	0	0

TABLE 1 – Triangle de Pascal à mettre en parallèle de la figure 2. On a représenté le triangle du bas vers le haut.

■ **Exemple 2 — Algorithme de calcul des termes de la suite Fibonacci.** On considère la suite de Fibonacci : $(u_n)_{n\in\mathbb{N}}$ telle que $u_0 = 0$, $u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$.

Pour calculer le u_n, on peut procéder de plusieurs manières différentes comme le montre les algorithmes 1 et 2. Cependant, ces deux approches n'ont pas la même efficacité. La première est une approche récursive multiple descendante. Les appels multiples montrent qu'on se trouve dans le cadre d'un problème pour lequel **les sous-problèmes ne sont pas indépendants**. Dans ce cas, l'algorithme 1 calcule inutilement plusieurs fois les mêmes termes et est inefficace. Par exemple, pour calculer u_4 selon cette approche on doit calculer $u_3 + u_2$. Mais le calcul de u_3 va lancer le calcul $u_2 + u_1$. On va donc calculer au moins deux fois u_2.

Cette approche aboutit à une complexité en $O(2^n)$.

L'algorithme 2 propose une version itérative ascendante dans l'ordre des termes : on calcule d'abord le premier terme puis le second et ainsi il n'y a pas de calculs redondants et la complexité en $O(n)$. Cette approche est dans l'esprit de la programmation dynamique : on cherche à calculer comme indiqué sur la figure 1 pour éviter les calculs redondants. On pourrait construire un graphe similaire à celui de la figure 2.

Algorithme 1 Fibonacci récursif (approche **descendante**)

```
1 : Fonction REC_FIBO(n)
2 :     si n = 0 ou n = 1 alors                          ▷ Condition d'arrêt
3 :         renvoyer n
4 :     sinon
5 :         renvoyer REC_FIBO(n-1) + REC_FIBO(n-2)       ▷ Appels multiples et chevauchements
```

Algorithme 2 Fibonacci itératif, sans calculs redondants (approche **ascendante**)

```
1 : Fonction ITE_FIBO(n)
2 :     u_0 ← 0 ; u_1 ← 1
3 :     pour i de 0 à n répéter
4 :         tmp ← u_0
5 :         u_0 ← u_1
6 :         u_1 ← tmp + u_1
7 :     renvoyer u_0
```

C Principes de la programmation dynamique

■ **Définition 1 — Principe d'optimalité de Bellman.** La solution optimale à un problème d'optimisation combinatoire présente la propriété suivante : quel que soit l'état initial et la décision initiale prise, les décisions qui restent à prendre pour construire une solution optimale forment une solution optimale par rapport à l'état qui résulte de la première décision [a].

a. PRINCIPLE OF OPTIMALITY. An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions[**bellman_theory_1954**].

■ **Définition 2 — Sous-structure optimale.** En informatique, un problème présente une sous-structure optimale si une solution optimale peut être construite à partir des solutions optimales à ses sous-problèmes.

■ **Définition 3 — Programmation dynamique.** La programmation dynamique est une méthode de construction des solutions optimales d'un problème par combinaison des solutions optimales de sous-problèmes. Pour cela, le problème considéré doit posséder une sous-structure optimale (cf. définition 2). Certaines combinaisons de solutions sont implicitement rejetées si elles appartiennent à un sous-ensemble qui n'est pas utile : afin d'être efficace, on ne construit que les solutions optimales des sous-problèmes utiles à la construction de la solution optimale.

Cette approche :

- considère un problème $\mathcal{P}$ à sous-structure optimale,
- décompose le problème $\mathcal{P}$ en sous-problèmes de taille moindre,
- construit une solution de $\mathcal{P}$ en ne résolvant un même sous-problème qu'une seule fois.

Le cadre de l'application de la programmation dynamique sont donc les problèmes d'optimisation combinatoire dont la sous-structure est optimale. Pour ce genre de problèmes, il y a de nombreuses solutions possibles [3]. Chaque solution possède une valeur propre que l'on peut quantifier. On cherche alors soit à la minimiser soit à la maximiser, dans tous les cas, on cherche au moins une valeur optimale.

3. Penser au problème du sac à dos par exemple.

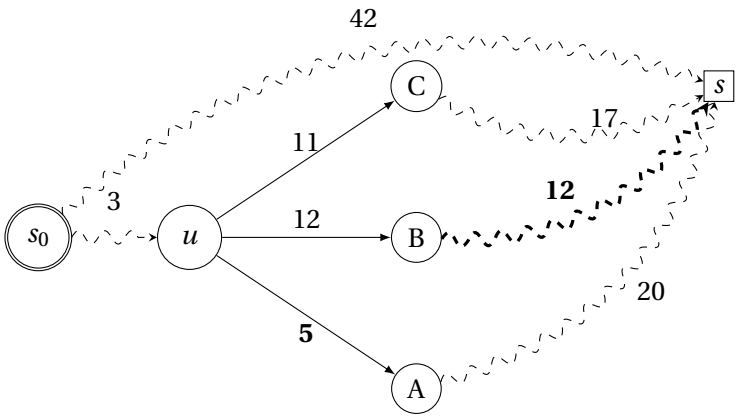

FIGURE 3 – Illustration du principe d'optimalité et de sous-structure optimale : trouver le plus court chemin dans un graphe orienté et pondéré. Les lignes droites sont des arcs. Les lignes ondulées indiquent les chemins connus dans le graphe : il faut imaginer qu'on n'a pas représenté tous les sommets. Les nombres représentent des distances ou les poids des arcs.

M **Méthode 1 — Algorithme de programmation dynamique, approche ascendante** Pour créer un algorithme de programmation dynamique avec une approche ascendante permettant de résoudre un problème à sous-structure optimale, il faut :

1. **Formuler récursivement** le problème en sous-problèmes,
2. Créer et initialiser un **tableau** de résolution,
3. Compléter ce tableau en calculant les solutions des sous-problèmes selon l'ordre de la récurrence trouvée pour ne les calculer qu'une seule fois.

D Principe d'optimalité et plus court chemin dans un graphe

Le principe d'optimalité et la notion de sous-structure optimale sont illustrés par la figure 3 qui représente le problème du plus court chemin dans un graphe. On peut exprimer le plus court chemin récursivement en fonction du début du chemin choisi et du plus court chemin dans le reste du graphe. Cela peut s'exprimer simplement comme suit :

Le plus court chemin de s_0 à s est le chemin le plus court à choisir parmi :

— le chemin ne passant pas par u, déjà découvert et dont la longueur vaut 42,

— le chemin qui passe par u et par A suivi par le chemin le plus court de A à s,

— le chemin qui passe par u et par B suivi par le chemin le plus court de B à s,

— le chemin qui passe par u et par C suivi par le chemin le plus court de C à s.

Il est nécessaire de considérer le plus court chemin de A à s ou de B à s, sinon la solution globale ne serait pas optimale. Le plus court chemin est une solution optimale et le plus court chemin

de A à s est un sous-problème. Donc, on exprime bien la solution optimale d'un problème en fonction des solutions optimales des sous-problèmes.

Le plus court chemin étant un problème à sous-structure optimale, on peut chercher à le résoudre par la programmation dynamique : cette approche est celle de l'algorithme de Bellman-Ford.

a Algorithme de Bellman-Ford

Soit $G = (S, A, w)$ un graphe orienté et pondéré ne possédant pas de circuits de poids négatif.

Si i représente **le nombre de sauts autorisés pour atteindre un sommet** et $d_i(v)$ la **distance du sommet de départ** s_0 au sommet v en effectuant i sauts, alors cela peut se traduit mathématiquement par :

$$d_i(v) = \begin{cases} 0 & \text{si } i = 0 \text{ et } v = s_0 \\ +\infty & \text{si } i = 0 \text{ et } v \neq s_0 \\ \min\left(d_{i-1}(v), \min\limits_{\text{arcs (u,v)}} (d_{i-1}(u) + w(u,v))\right) & \text{sinon} \end{cases} \tag{2}$$

Le problème ainsi posé étant à sous-structure optimale, l'algorithme de Bellmann-Ford 3 résout ce problème en utilisant la programmation dynamique. L'ordre du graphe n étant donné, cet algorithme nécessite $n-1$ itérations pour converger car le chemin le plus court passe par au plus $n-1$ sommets. À chaque itération, pour chaque arête (u, v), il choisit la distance la plus courte pour atteindre v.

Algorithme 3 Algorithme de Bellman-Ford, plus courts chemins à partir d'un sommet donné

```
1 : Fonction BELLMAN_FORD(G = (S, A, w), s_0)
2 :     n ← ordre de G                          ▷ n est le nombre de sommets de G
3 :     d ← un tableau de dimension (n, n+1)    ▷ distances au sommet a
4 :     d[0,:] ← w(s_0, s)   ▷ Initialisation : w(s_0, s) = +∞ si s n'est pas voisin de s_0, 0 si s = s_0
5 :     pour i de 1 à n − 1 répéter
6 :         pour (u, v) = a ∈ A répéter          ▷ Tester un saut de plus via l'arête (u, v)
7 :             si d[i − 1, v] > d[i − 1, u] + w(u, v) alors   ▷ Si plus court par u
8 :                 d[i, v] ← d[i − 1, u] + w(u, v)   ▷ Mises à jour de la distance de s_0 à v en i sauts
9 :     renvoyer d[n − 1, :]                    ▷ On renvoie les distances de s_0 à tous les sommets
```

R La complexité de l'algorithme de Bellman-Ford est en $O(nm)$, pour un graphe d'ordre n possédant m arêtes.

■ **Exemple 3 — Application de l'algorithme de Bellman-Ford.** On se propose d'appliquer l'algorithme 3 au graphe pondéré et orienté représenté sur la figure 4. Ce graphe contient des pondérations négatives mais pas de cycles à pondération négative. Le tableau 2 re-

présente les distances successivement trouvées à chaque itération. La complexité de l'algorithme est en $O(nm)$ si n est l'ordre du graphe et m sa taille.

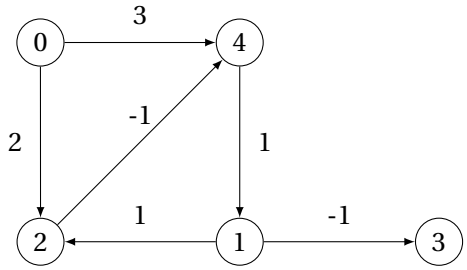

FIGURE 4 – Graphe orienté et pondéré pour application de l'algorithme de Bellman-Ford (sans circuit à pondération négative)

i \ v	0	1	2	3	4
4	0	2	2	1	1
3	0	2	2	1	1
2	0	2	2	3	1
1	0	4	2	$+\infty$	1
0	0	$+\infty$	2	$+\infty$	3

TABLE 2 – Tableau lié à l'application de l'algorithme de Bellman-Ford au départ du sommet 0 du graphe de la figure 4. Application de la formule de récurrence 2 : i représente le nombre de sauts autorisés, v les sommets. Complétion du bas vers le haut.

Il est important de souligner que cet algorithme s'applique uniquement à des **graphes pondérés et orientés** dont les pondérations peuvent être négatives mais **sans cycles de longueur négative** [**bellman_routing_1958**, **ford_jr_network_1956**, **moore_shortest_1959**].

R Les poids négatifs peuvent représenter des transferts de flux (énergie ou chaleur en physique-chimie, argent en économie) et sont donc très courants.

R Les cycles de poids négatif ne peuvent pas permettre de définir une distance minimale : à chaque itération du cycle, la distance diminue.

R Cet algorithme ne s'applique pas à des graphes non orientés pour la raison suivante : les arêtes d'un graphe non orienté sont des cycles car la relation est dans les deux sens. Donc chaque arête de poids négatif est un cycle de poids négatif.

■ **Exemple 4 — Protocole de routage RIP.** Le protocole de routage RIP utilise l'algorithme de Bellman-Ford pour trouver les plus cours chemins dans un réseau de routeur. Il est moins adapté que OSPF pour les grands réseaux à cause de la lenteur de la convergence : pour être sûr de disposer des chemins les plus cours, il faut obligatoirement opérer les $n-1$ itérations de l'algorithme.

b Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall [**floyd_algorithm_1962**, **roy_transitivite_1959**, **warshall_theorem_1962**] est l'application de la programmation dynamique[4] à la recherche de **l'existence d'un chemin entre toutes les paires de sommets d'un graphe orienté et pondéré**. Les distances trouvées sont les plus courtes. **Les pondérations du graphe peuvent être négatives mais on exclut tout circuit de poids strictement négatif**.

Soit un graphe orienté et pondéré $G = (S, A, w)$. G peut être modélisé par une matrice d'adjacence M

$$\forall i, j \in [\![0, |S|-1]\!], M = \begin{cases} w(i,j) & \text{si } (i,j) \in A \\ +\infty & \text{si } (i,j) \notin A \\ 0 & \text{si } i = j \end{cases} \tag{3}$$

Un exemple de graphe associé à la matrice d'adjacence :

$$M_{\text{init}} = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & -3 & 0 \end{pmatrix} \tag{4}$$

est donné sur la figure 5. Sur cet exemple, le chemin le plus court de 0 à 2 vaut -2 et passe par 3.

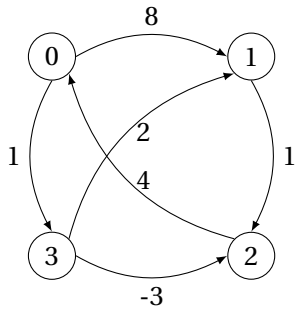

FIGURE 5 – Exemple de graphe orienté et pondéré pour expliquer le concept de matrice d'adjacence.

Chaque étape p de l'algorithme de Floyd-Warshall ne considère que les chemins possibles n'utilisant que les p premiers sommets. À l'étape p, on associe une matrice M_p qui contient

4. cf. programme de seconde année

la longueur des chemins les plus courts d'un sommet à un autre passant par des sommets de l'ensemble $\{v_0, v_1, \ldots, v_{p-1}\}$. On construit ainsi une suite de matrice finie $\left(M_p\right)_{p\in[\![0,n]\!]}$ et on initialise la matrice M avec avec M_{init}.

Supposons qu'on dispose de M_{p-1}. Considérons un chemin $\mathcal{C}$ entre v_i et v_j dont la longueur est minimale et dont les sommets intermédiaires sont dans $\{v_0, v_1, \ldots, v_{p-2}\}$, $p \leqslant n$. Pour un tel chemin :

— soit le plus court chemine passe par v_{p-1}. Dans ce cas, $\mathcal{C}$ est la réunion de deux chemins dont les sommets sont dans $\{v_0, v_1, \ldots, v_{p-1}\}$: celui de v_i à v_{p-1} et celui de v_{p-1} à v_j.

— soit le plus court chemin ne passe pas par v_{p-1}.

Entre ces deux chemins, on choisira le chemin le plus court.

R Le chemin le plus court est au maximum de longueur n, sinon, on repasserait nécessairement par un sommet déjà visité (principe des tiroirs).

On peut traduire notre explication ci-dessus par la relation de récurrence suivante :

$$\forall p \in [\![1,n]\!], \forall i,j \in [\![0,n-1]\!], M_p(i,j) = \min\left(M_{p-1}(i,j), M_{p-1}(i,p-1) + M_{p-1}(p-1,j)\right) \quad (5)$$

Pour $p = 0$, on pose $M_0 = M_{\text{init}}$.

L'algorithme de Floyd-Warshall 4 est un bel exemple de programmation dynamique. Sa complexité temporelle est en $O(n^3)$. Il peut être programmé en place.

Algorithme 4 Algorithme de Floyd-Warshall, plus courts chemins entre toutes les paires de sommet

```
1 : Fonction FLOYD_WARSHALL(G = (S, A, w))
2 :     M ← la matrice d'adjacence de G                ▷ Correspond à M_0, p = 0
3 :     pour p de 1 à |S| répéter                      ▷ Nombre de sauts
4 :         pour i de 0 à |S| − 1 répéter              ▷ Sommet de départ
5 :             pour j de 0 à |S| − 1 répéter          ▷ Sommet d'arrivée
6 :                 M_p(i,j) = min(M_{p−1}(i,j), M_{p−1}(i,p−1) + M_{p−1}(p−1,j))
7 :     renvoyer M
```

R Cet algorithme ressemble à Bellman-Ford mais avec une vision globale, à l'échelle du graphe tout entier, pas uniquement par rapport à un sommet de départ. Il ne considère, à chaque itération, que les chemins possibles passant par les sommets d'indice au plus p. C'est pourquoi sa complexité est en $O(n^3)$.

■ **Exemple 5 — Application de l'algorithme de Floyd-Warshall.** Si on applique l'algorithme

au graphe de la figure 5, alors on obtient la série de matrices suivantes :

$$M_0 = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & -3 & 0 \end{pmatrix} \tag{6}$$

$$M_1 = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & 12 & 0 & 5 \\ +\infty & 2 & -3 & 0 \end{pmatrix} \tag{7}$$

$$M_2 = \begin{pmatrix} 0 & 8 & 9 & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & 12 & 0 & 5 \\ +\infty & 2 & -3 & 0 \end{pmatrix} \tag{8}$$

$$M_3 = \begin{pmatrix} 0 & 3 & -2 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 1 & 2 & -3 & 0 \end{pmatrix} \tag{9}$$

E Le retour du sac à dos

a Position du problème

On cherche à remplir un sac à dos comme indiqué sur la figure 6. Chaque objet que l'on peut insérer dans le sac est **insécable**[5] et possède une valeur et un poids connus. On cherche à maximiser la valeur totale emportée dans la sac à dos tout en limitant[6] le poids à π.

On a vu au chapitre **??** que l'approche gloutonne ne donnait pas toujours le résultat optimal. On se propose donc de résoudre le problème par la programmation dynamique en appliquant la méthode 1.

b Modélisation du problème

Soit un ensemble $\mathcal{O}_n = \{o_1, o_2, \ldots, o_n\}$ de n objets de valeurs $v_1, v_2, \ldots, v_n$ et de poids respectifs $p_1, p_2, \ldots, p_n$. Soit un sac à dos n'admettant pas un poids emporté supérieur à π. On note également qu'on peut mettre au plus n objets dans le sac.

Les objets sont rangés dans une liste et dans un ordre quelconque. Ils sont indicés par i variant de 1 à n. Un objet o_i possède une valeur v_i et un pèse p_i.

Avec ces notations, on peut formuler le problème du sac à dos comme suit.

5. Soit on le met dans le sac, soit on ne le met pas. Mais on ne peut pas en mettre qu'une partie.
6. On accepte un poids total inférieur ou égal à π.

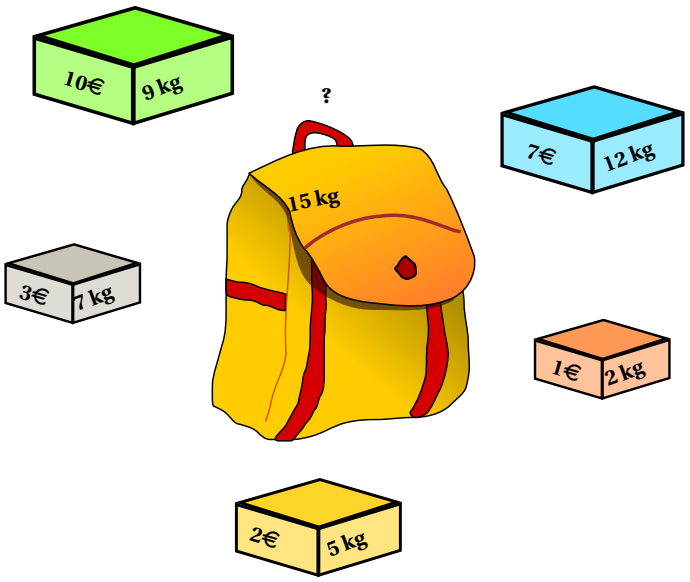

FIGURE 6 – Illustration du problème du sac à dos (d'après Wikipedia). On a cinq objets de poids 9, 12, 2, 7 et 5 kg et de valeur 10, 7, 1, 3 et 2 €. Le poids total admissible dans le sac est 15kg.

■ **Définition 4 — Problème du sac à dos.** Comment remplir un sac à dos en maximisant la valeur totale emportée V tout en ne dépassant pas le poids maximal π admissible par le sac à dos.

Formellement, comment maximiser $V = \sum_{o_i \in B} v_i$ en respectant la contrainte $\sum_{o_i \in B} p_i \leqslant \pi$ où B est l'ensemble des objets emportés dans le sac ?

On note[a] le problème du sac à dos $\mathrm{KP}(n,\pi)$ et une solution optimale à ce problème $S(n,\pi)$.

a. en anglais, ce problème est nommé Knapsack Problem, d'où le KP.

c Formulation récursive du problème en sous-problèmes non indépendants

Pour chaque objet o_i, si $p_i \leqslant \pi$, on peut le mettre dans le sac. La formulation récursive s'énonce alors ainsi :

— Soit l'objet o_i fait partie d'une solution optimale. Alors la fonction à maximiser vaut la valeur de l'objet o_i plus la valeur maximale atteignable avec les $n-1$ objets restants, sachant qu'on ne peut plus mettre que $\pi - p_i$ kg dans le sac.

— Soit l'objet o_i ne fait pas partie d'une solution optimale. Alors la fonction à maximiser vaut la valeur maximale atteignable avec les $n-1$ objets restants une fois cet objet o_i écarté. On peut toujours mettre π kg dans le sac.

Formellement, on exprime cette récursivité ainsi :

$$S(n,\pi) = \begin{cases} 0 & \text{si } n = 0 \text{ ou si } \pi = 0 \\ \max\big(v_n + S(n-1,\pi-p_n), S(n-1,\pi)\big) & \text{si } p_n \leqslant \pi \\ S(n-1,\pi) & \text{sinon} \end{cases} \tag{10}$$

Cette formulation prouve que **le problème du sac à dos possède une sous-structure optimale et que les problèmes se chevauchent** : pour un poids π maximal donné, **calculer une solution optimale de KP** (n,π) **nécessite de savoir calculer une solution optimale pour les** $n-1$ **premiers objets de la liste et pour des poids** π **et** $\pi - p_i$. Schématiquement, on peut représenter cette démarche comme sur la figure 7.

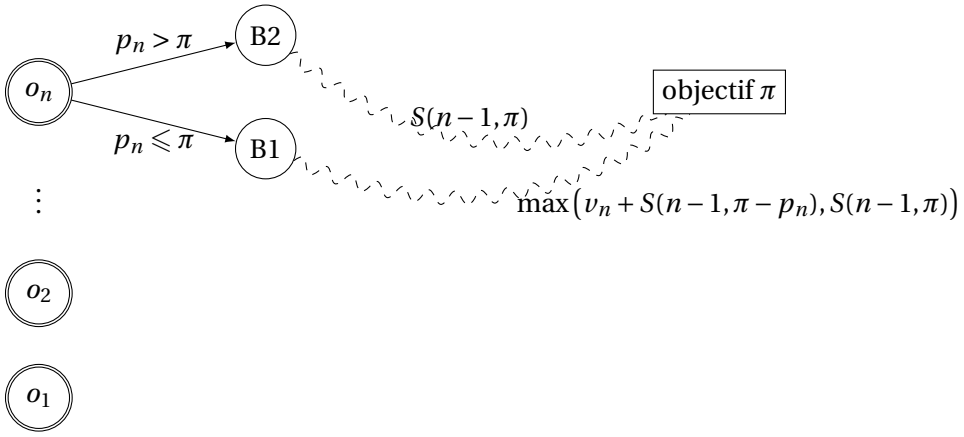

FIGURE 7 – Formulation récursive du problème du sac à dos.

R Attention au sens des notations : $S(i,P)$ est une solution au problème KP (i,P). Pour ce problème, on ne peut prendre que les i premiers objets de la liste et le poids maximum admissible est P. Si on considère $S(i-1,P-p_i)$, alors on ne peut prendre que les $i-1$ premiers objets de la liste et le poids maximal admissible est $P-p_i$. Les objets sont rangés dans un ordre quelconque.

d Création et initialisation du tableau de résolution

On cherche donc maintenant à créer un tableau à double entrée qui recense toutes les solutions optimales nécessaires à la résolution du problème KP (n,π). Ce tableau a pour dimension $(n+1,\pi+1)$:

- le nombre i d'objets dans le sac d'un côté à valeur dans $[\![0,n]\!]$. La valeur pour $i=0$ est nulle, on ne prend pas d'objet.
- les poids P atteignables de l'autre à valeur dans $[\![0,\pi]\!]$. La valeur pour $P=0$ est nulle, on ne prend pas d'objet.

La valeur d'une case du tableau est $S(i,P)$.

Indice i de l'objet	1	2	3	4	5
valeur (€)	10	7	1	3	2
poids (kg)	9	12	2	7	5

TABLE 3 – Synthèse des informations relatives au problème de la figure 6.

Indice i																
5	0	0	1	1	1	2	2	3	3	10	10	11	11	11	12	12
4	0	0	1	1	1	1	1	3	3	10	10	11	11	11	11	11
3	0	0	1	1	1	1	1	1	1	10	10	11	11	11	11	11
2	0	0	0	0	0	0	0	0	0	10	10	10	10	10	10	10
1	0	0	0	0	0	0	0	0	0	10	10	10	10	10	10	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Poids (kg) →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

TABLE 4 – Tableau de résolution du sac à dos dans le cas de la figure 6 donnant les valeurs de $S(i,P)$, avec $i \in [\![0,5]\!]$ et $P \in [\![0,15]\!]$.

e Complétion du tableau par résolution ascendante

On considère la liste d'objets décrite sur le tableau 3 et qui correspond au problème décrit sur la figure 6. L'ordre des objets est **arbitraire**. Le résultat du calcul est donné sur le tableau 4. On a construit ce tableau du bas vers le haut en suivant l'algorithme 5.

Naturellement, sur l'exemple donné sur la figure 6, il est possible de calculer à la main les valeurs du tableau. Ce n'est guère le cas dans des situations réalistes, c'est pourquoi il faut maintenant écrire l'algorithme qui va permettre de compléter ce tableau dans l'ordre et ainsi de résoudre notre problème.

R Pour bien comprendre, il peut être utile de reproduire la figure 2 pour l'exemple du sac à dos KP(5,7) par exemple!

R Le tableau de résolution est formé en général de $n+1$ lignes et $m+1$ colonnes. Le sens de la complétion du tableau dépend de la récurrence trouvée. Si les expressions trouvées sont du type :

- $S(i-1, j-2)$, il faut envisager la complétion du tableau de bas en haut et de gauche à droite.
- $S(i+1, j+2)$, il faut envisager la complétion du tableau de haut en bas et de droite à gauche.

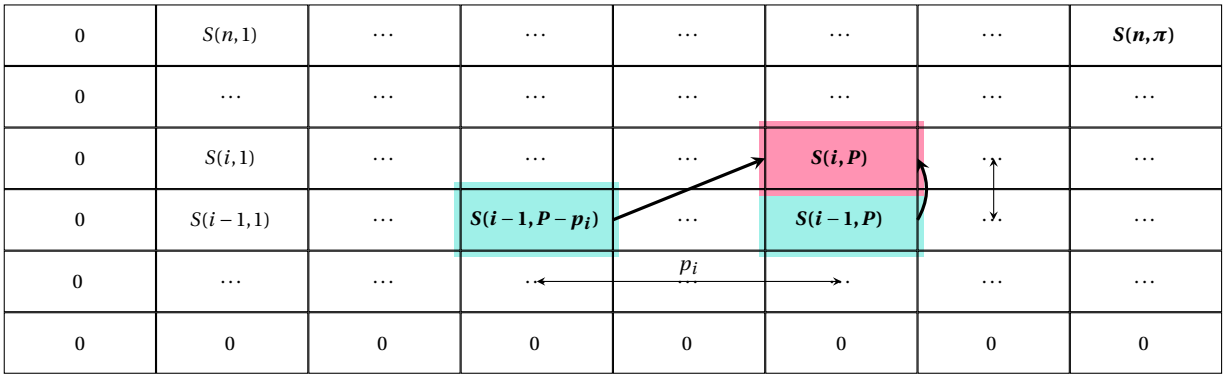

FIGURE 8 – Schéma de remplissage du tableau pour le problème KP(n,π). Le poids se trouve sur l'axe horizontal et le nombre d'objets sur l'axe vertical. Pour calculer $S(i,P)$ on a besoin de $S(i-1,P)$ et de $S(i-1,P-p_i)$.

F Programmation dynamique itérative (approche ascendante)

L'algorithme 5 donne la procédure de résolution de KP(n,π) par programmation dynamique, de bas en haut et de gauche à droite et de manière itérative. Aucun calcul redondant n'est effectué.

Algorithme 5 KP(n,π) par programmation dynamique

```
1 : Fonction KP_DP(p, v, π, n)                    ▷ p la liste de poids, v celle des valeurs
2 :     S ← un tableau d'entiers de taille (n+1, π+1)
3 :     pour i de 0 à n répéter                                          ▷ de bas en haut
4 :         pour P de 0 à π répéter                                      ▷ de gauche à droite
5 :             si i = 0 ou P = 0 alors
6 :                 S[i,P] ← 0
7 :             sinon si p_i ⩽ P alors
8 :                 S[i,P] ← max(v_i + S[i−1, P−p_i], S[i−1,P])
9 :             sinon
10 :                S[i,P] ← S[i−1,P]
11 :    renvoyer S[n,π]
```

Les complexités temporelle et spatiale de l'algorithme 5 sont en $O(n\pi)$.

G Programmation dynamique récursive (approche descendante)

Il est possible de résoudre le problème du sac à dos de manière récursive comme le montre l'algorithme 6. Néanmoins, comme les sous-problèmes se chevauchent, de nombreux calculs redondants sont effectués. Pour des valeurs importantes de n et π, cet algorithme est totale-

ment inefficace.

Algorithme 6 KP(n,π) par programmation récursive brute

```
1 : Fonction KP_REC(p,v, π, n)                    ▷ p la liste de poids, v celle des valeurs
2 :     si i = 0 ou P = 0 alors
3 :         renvoyer 0
4 :     sinon si p[i] ⩽ P alors
5 :         renvoyer max(v_n + KP_REC(p,v, π − p_n, n − 1),KP_REC(p,v, π, n − 1))
6 :     sinon
7 :         renvoyer KP_REC(p,v, π, n − 1)
```

■ **Définition 5 — Mémoïsation.** La mémoïsation est une technique de mise en mémoire de résultats intermédiaires afin de ne pas les recalculer.

Pour résoudre les problèmes de l'algorithme 6, les calculs intermédiaires sont stockés dans une structure de données, typiquement un tableau ou un dictionnaire. Avant chaque appel récursif, l'algorithme vérifie si le calcul à faire récursivement a déjà été effectué. Si c'est le cas, la solution stockée dans le tableau est utilisée. Sinon la récursivité s'exécute.

L'algorithme 7 donne la procédure de résolution de KP(n,π) en utilisant la mémoïsation. Cette technique récursive est considérée comme une implémentation possible de la programmation dynamique : les langages contemporains permettent même de l'automatiser (cf. décorateur `@lru_cache()` de la bibliothèque `functools` en Python).

Algorithme 7 KP(n,π) par programmation dynamique et mémoïsation

```
1 : Fonction KP_MEM(p,v, π, n,S)                  ▷ S est un tableau d'entiers de taille (n + 1, π + 1)
2 :     si i = 0 ou P = 0 alors
3 :         renvoyer 0
4 :     sinon
5 :         si la solution S[n, P] a déjà été calculée alors
6 :             renvoyer S[n, P]
7 :         sinon si p_n ⩽ π alors
8 :             S[n, P] ← max(v_n + KP_MEM(p,v, π − p_n, n − 1, S),KP_MEM(p,v, π, n − 1, S))
9 :             renvoyer S[n, P]
10 :        sinon
11 :            S[n, P] ← KP_MEM(p,v, π, n − 1)
12 :            renvoyer renvoyer S[n, P]
```