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.

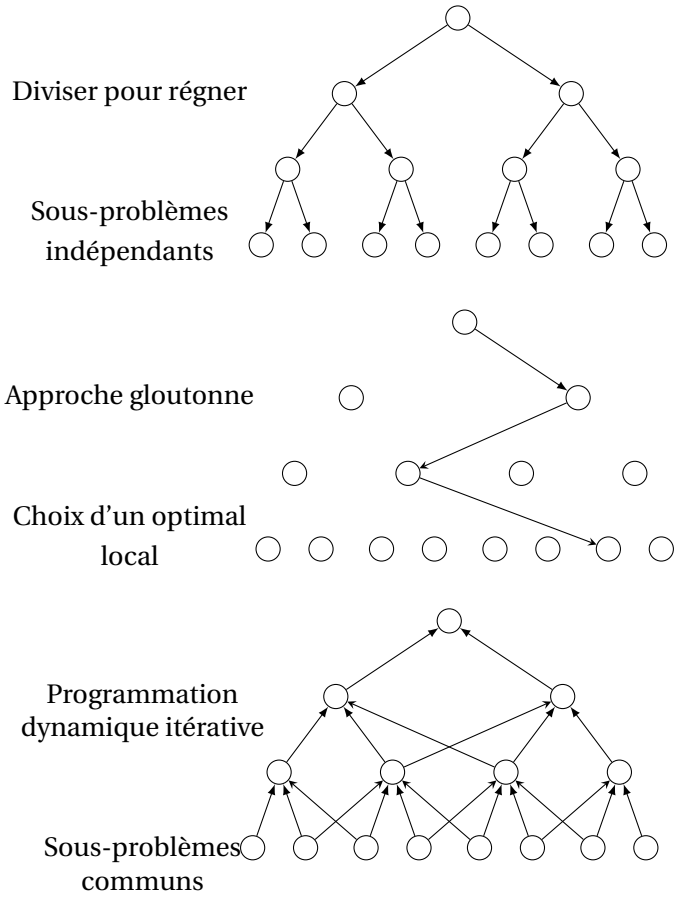

FIGURE 1 – Schématisation des différentes approches des algorithmes de décomposition d'un problème en sous-problèmes : diviser pour régner, approche gloutonne et programmation dynamique itérative.

B Principes de la programmation dynamique

■ **Définition 1 — Programmation dynamique.** La programmation dynamique est une méthode de construction des solutions optimales d'un problème par combinaison des solutions optimales de sous-problèmes. Certaines combinaisons sont implicitement rejetées si elles appartiennent à un sous-ensemble qui n'est pas intéressant : on ne construit que les solutions optimales des sous-problèmes utiles à la construction de la solution optimale. Cette approche :

- décompose le problème $\mathcal{P}$ en sous-problèmes de taille moindre,
- ne résout un sous-problème qu'une seule fois.

En général, le cadre de l'application de la programmation dynamique sont 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.

M **Méthode 1 — Développement d'un algorithme de programmation dynamique** Pour créer un algorithme de programmation dynamique répondant à un problème, il faut :

1. Formuler récursivement le problème en sous-problèmes ordonnés,
2. Créer et initialiser un tableau de résolution,
3. Compléter ce tableau du bas vers le haut en calculant les solutions des sous-problèmes dans l'ordre croissant de la récurrence trouvée.

■ **Définition 2 — 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 3 — Sous-structure optimale.** En informatique, un problème présente une sous-structure optimale si une solution optimale peut être construire à partir des solutions optimales à ses sous-problèmes.

Le principe d'optimalité et la notion de sous-structure optimale sont illustrés par la figure 2 qui représente le problème du plus court chemin dans un graphe. Résoudre ce problème via la programmation dynamique suppose qu'on a un problème à sous-structure optimale, ce qui est le cas.

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

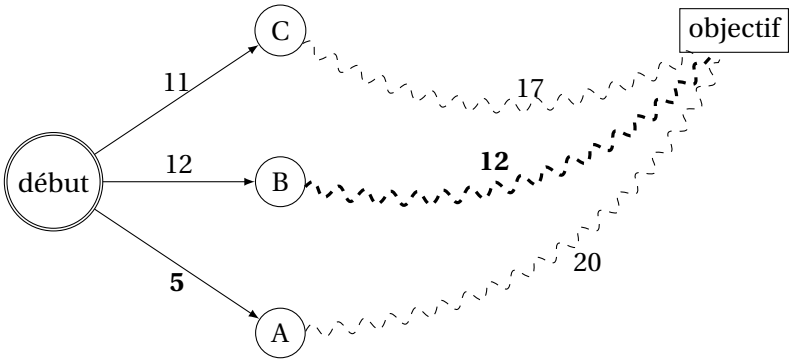

FIGURE 2 – Illustration du principe d'optimalité et de sous-structure optimale : trouver le plus court chemin dans ce graphe. Les nombres représentent la longueur d'un chemin. Les lignes droites sont des arrêtes. Les lignes ondulées indiquent les plus courts chemins dans le graphe : il faut imaginer qu'on n'a pas représenté tous les sommets.

On peut exprimer le plus court chemin récursivement en fonction du premier chemin choisi et du reste du graphe. Cela peut s'écrire formellement mathématiquement ou, plus simplement, comme suit :

Le plus court chemin est le chemin le plus court à choisir parmi :

- le chemin qui passe par A suivi par le chemin le plus court de A à l'objectif,
- le chemin qui passe par B suivi par le chemin le plus court de B à l'objectif,
- le chemin qui passe par C suivi par le chemin le plus court de C à l'objectif.

La résolution du problème amènera à choisir le chemin qui passe par B.

C Exemples simples de programmation dynamique

■ **Exemple 1 — Construction des solutions du 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). On peut considérer ce triangle comme le tableau de résolution du problème du calcul de $\binom{n}{k}$ par une méthode de programmation dynamique (cf. figure 3).

■ **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 $u_{n+2} = u_{n+1} + u_n$.

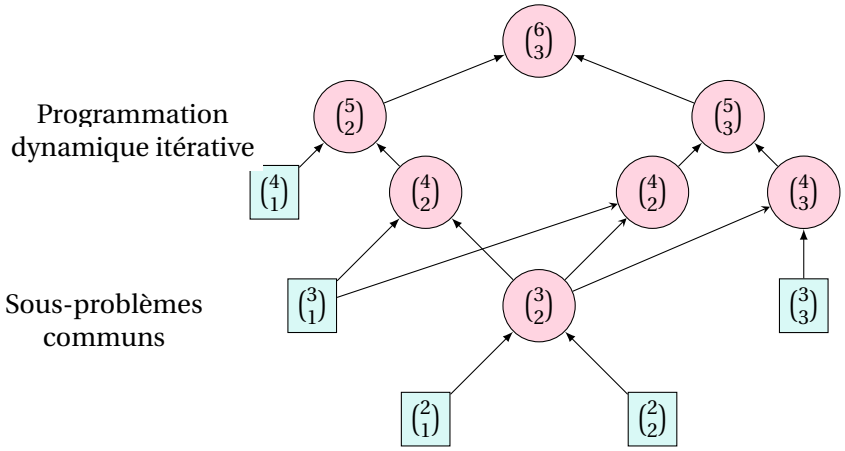

FIGURE 3 – Programmation dynamique du calcul de $\binom{6}{3}$.

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 : valeurs de $\binom{n}{k}$. En couleur apparaissent les éléments nécessaires pour calculer $\binom{6}{3}$ qu'il faut mettre en parallèle de la figure 3. On a représenter le triangle du bas vers le haut afin de montrer le lien avec la programmation dynamique.

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 deux fois u_2.

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. Cette approche est dans l'esprit de la programmation dynamique : on cherche à calculer du bas vers le haut comme indiqué sur la figure 1 pour éviter les calculs redondants. On pourrait construire un graphe similaire à celui de la figure 3.

Algorithme 1 Fibonacci récursif

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

Algorithme 2 Fibonacci itératif, sans calculs redondants.

```
1 : Fonction ITE_FIBO(n)
2 :     si n = 0 alors
3 :         renvoyer 0
4 :     sinon si n =1 alors
5 :         renvoyer 1
6 :     sinon
7 :         u_0 ← 0
8 :         u_1 ← 1
9 :         pour i de 2 à n répéter
10 :            tmp ← u_0
11 :            u_0 ← u_1
12 :            u_1 ← tmp + u_1
13 :    renvoyer u_1
```

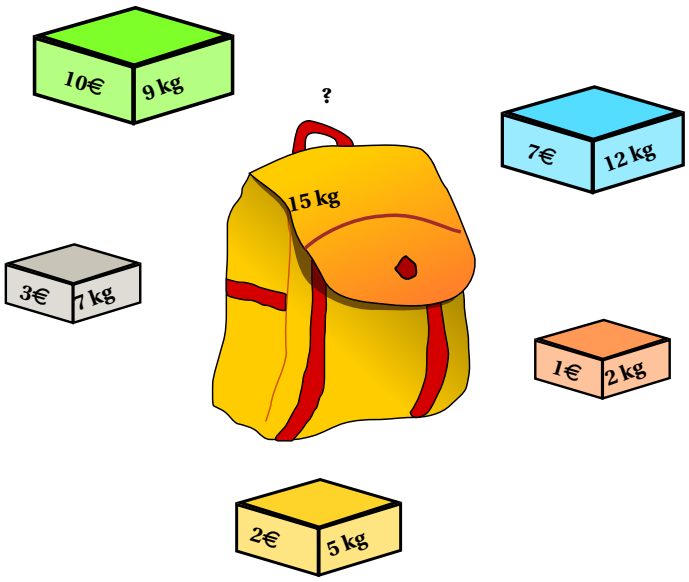

FIGURE 4 – Illustration du problème du sac à dos (d'après Wikipedia). On a cinq objets de poids 9, 12, 2, 7 et 5 kg et de valeur 10, 7, 1, 3 et 2 €. Le poids total admissible dans le sac est 15kg.

D Le retour du sac à dos

a Position du problème

On cherche à remplir un sac à dos comme indiqué sur la figure 4. Chaque objet que l'on peut insérer dans le sac est **insécable**[4] et possède une valeur et un poids connus. On cherche à maximiser la valeur totale emportée dans la sac à dos tout en limitant[5] 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.

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

■ **Définition 4 — Problème du sac à dos.** Comment remplir un sac à dos en maximisant la valeur totale emportée $V = \sum_{i=1}^{n} v_i$ tout en ne dépassant pas le poids maximal admissible par le sac à dos, c'est à dire en respectant la contrainte $\sum_{i=1}^{n} p_i \leqslant \pi$? 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 :

$$S(n,\pi) = \begin{cases} 0 & \text{si } n = 0 \text{ ou si } \pi = 0 \\ \max\big(v_i + S(n-1,\pi-p_i), S(n-1,\pi)\big) & \text{si } p_i \leqslant \pi \\ S(n-1,\pi) & \text{sinon} \end{cases} \tag{2}$$

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** $(\mathcal{O}_n,\pi)$ **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 5.

R Attention au sens des notations : $S(i,P)$ est une solution au problème $\mathrm{KP}(i,P)$. 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 $\mathrm{KP}(\mathcal{O}_n,\pi)$. 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.

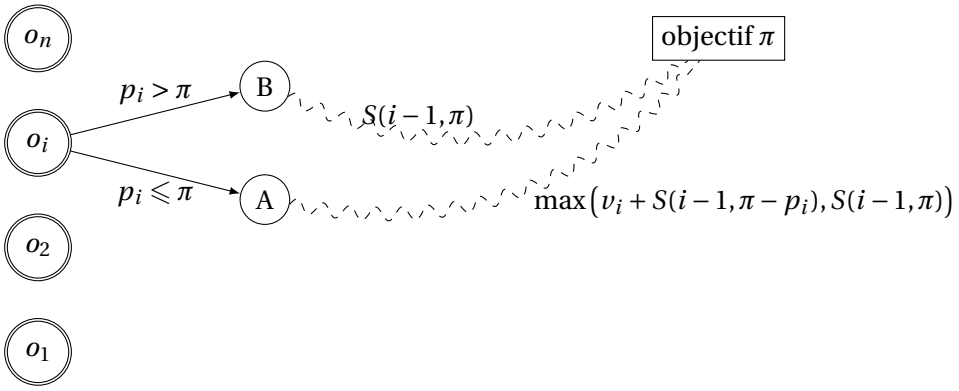

FIGURE 5 – Formulation récursive du problème du sac à dos.

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

TABLE 2 – Synthèse des informations relatives au problème de la figure 4.

- 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 $S(i,P)$.

e Complétion du tableau par résolution dans l'ordre croissant des sous-problèmes

On considère la liste d'objets décrite sur le tableau 2 et qui correspond au problème décrit sur la figure 4. L'ordre des objets est **arbitraire**. Le résultat du calcul est donné sur le tableau 3. On a construit ce tableau du bas vers le haut en suivant l'algorithme 3.

Naturellement, sur l'exemple donné sur la figure 4, 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 3 pour l'exemple du sac à dos KP(5,7) par exemple!

E Programmation dynamique itérative

L'algorithme 3 donne la procédure de résolution de KP(n,π) par programmation dynamique, de bas en haut et de manière itérative. Aucun calcul redondant n'est effectué puisqu'on

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

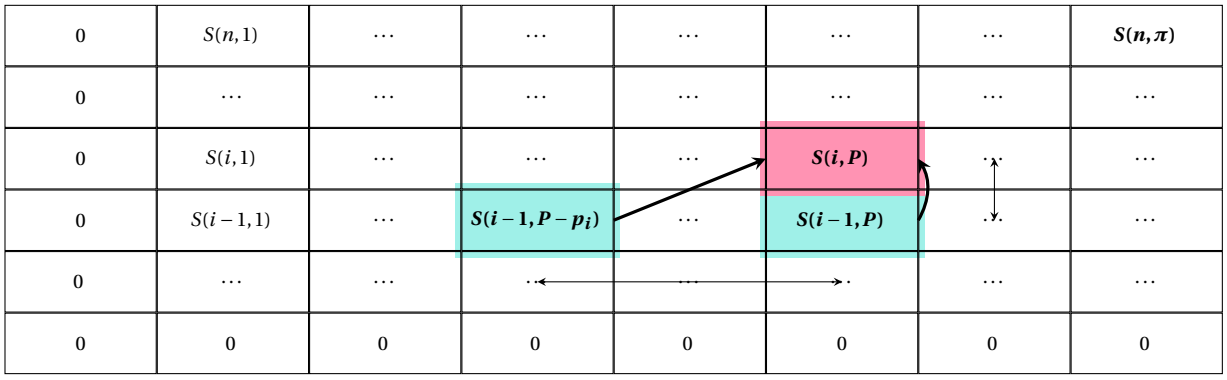

FIGURE 6 – Schéma de remplissage du tableau pour le problème KP(n,π). Pour calculer $S(i,P)$ on a besoin de $S(i-1,P)$ et de $S(i-1,P-p_i)$.

complète le tableau de bas en haut.

Algorithme 3 KP(n,π) par programmation dynamique, de bas en haut

```
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                                      ▷ d bas en haut
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 3 sont en $O(n\pi)$.

F Programmation dynamique récursive : mémoïsation

Il est possible de résoudre le problème du sac à dos de manière récursive comme le montre l'algorithme 4. Néanmoins, comme les sous-problèmes se chevauchent, de nombreux calculs redondants sont effectués. Pour des valeurs importantes de n et π, cet algorithme est totalement inefficace.

Algorithme 4 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 approche récursive de la programmation dynamique qui stocke les résultats intermédiaires dans un tableau et qui, avant chaque appel récursif, 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 5 donne la procédure de résolution de KP(n,π) par mémoïsation. Cette technique récursive est une approche de la programmation dynamique récursive.

Algorithme 5 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]
```

R La mémoïsation est une technique qui se prête bien à l'usage de dictionnaires : on peut stocker chaque solution déjà calculée dans un dictionnaire dont la clef est le couple (i,P) et la valeur la solution au problème KP(i,P). L'accès à la solution est rapide, en $O(1)$.