Cours 09 - Programmation Dynamique

MPSI - Prytanée National Militaire

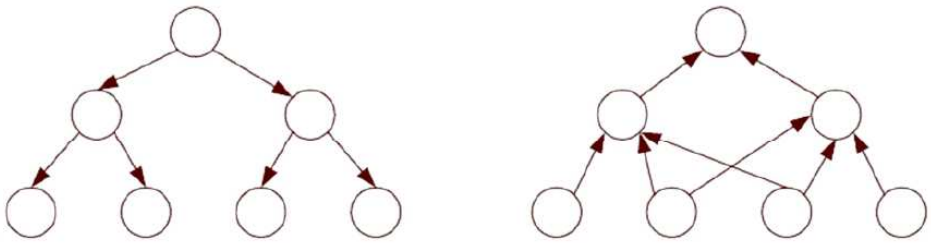
Pascal Delahaye

12 juin 2017

1 Introduction

La programmation dynamique est un paradigme de conception itératif adapté aux fonctions récursives qui permet d'améliorer leur complexité. Ce concept a été introduit par Bellman, dans les années 50, pour résoudre typiquement des problèmes d'optimisation. De nos jours, l'application de cette méthode ne se limite plus à ce type de problèmes. Ainsi :

- Comme dans le cas d'un algorithme récursif, la programmation dynamique s'applique lorsque la solution d'un problème peut s'exprimer en fonction des solutions de sous-problèmes.
- Au lieu de passer par une programmation de type récursive, la programmation dynamique est de type itérative.
- La programmation dynamique évite, contrairement aux programmes récursifs la répétition éventuelle de calculs identiques. Il en résulte un gain significatif en terme de complexité temporelle.



Programmation Récursive — Programmation Dynamique

- Comme l'illustre le schéma ci-dessus, les calculs dans un programme récursif se font de haut en bas, tandis qu'en programmation dynamique ils s'effectuent de bas en haut (algorithme de type "bottom-up") : on commence par résoudre les plus petits sous-problèmes. En "combinant" leur solution, on obtient les solutions des sous-problèmes de plus en plus grands. Cela est rendu possible en sauvegardant tous les résultats obtenus pour les sous-problèmes dans un tableau à une ou plusieurs dimensions.

`Exemple 1.` Premiers exemples.

1. La suite de Fibonacci : $\begin{cases} F_0 = 0 \\ F_1 = 1 \end{cases}$ et $\forall n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n$.

CAML
```
let Fib n = let V = make_vect n 0 in
            V.(0) <- 0;
            V.(1) <- 1;
            for k = 2 to (n-1) do V.(k) <- V.(k-1) + V.(k-2)
                               done;
            V.(n-1);;
```

Dans ce programme, les sous-problèmes correspondent au calcul des valeurs $F_0,\ F_1, \ldots,\ F_{n-1}$.
Ces valeurs sont stockées dans un tableau ligne.
Il s'agit bien d'un algorithme de type "bottom-up" puisque l'on commence par calculer F_0 puis F_1, etc, pour remonter progressivement vers la valeur demandée F_n.

2. Le calcul des coefficients binomiaux $\binom{n}{k}$.

CAML
```
let binom k n = let T = make_matrix (n+1) (n+1) 0 in
                for i = 0 to n do T.(i).(0) <- 1;
                                     T.(i).(i) <- 1
                                  done;
                for i = 2 to n do
                    for j = 1 to (min k (i-1)) do
                        T.(i).(j) <- T.(i-1).(j-1) + T.(i-1).(j)
                                     done;
                                  done;
T.(n).(k);;
```

Dans ce programme, les sous-problèmes correspondent au calcul des valeurs $\binom{0}{0}$, $\binom{0}{1}$, $\binom{1}{1}$ etc.
Ces valeurs sont stockées dans une matrice (dimension 2).
Il s'agit bien d'un algorithme de type "bottom-up" puisque l'on commence par $\binom{0}{0}$ puis $\binom{0}{1}$ etc..., pour remonter progressivement vers la valeur demandée $\binom{k}{n}$.

Exercice : 1

Le nombre de surjections d'un ensemble de n éléments vers un ensemble à p éléments est donné par la formule de récurrence :

$$S_n^p = p(S_{n-1}^{p-1} + S_{n-1}^p)$$

Programmer le calcul de S_n^p à l'aide d'un algorithme de programmation dynamique.

2 Quand et comment utiliser la méthode de programmation dynamique ?

Dans le cas d'un programme récursif, il est légitime d'envisager d'appliquer un mode de programmation dynamique lorsque l'on se rend compte que la solution d'un même sous-problème est calculée plusieurs fois. C'est le cas en particulier du calcul récursif des coefficients binomiaux ou des termes de la suite de Fibonacci.

Le nombre de paramètres entiers intervenant dans la formule de récursivité conduit naturellement à **la définition de la table** qui peut être de dimension 1 (cas d'un seul paramètre entier : suite de Fibonacci), de dimension 2 (cas de 2 paramètres entiers : coefficients binomiaux) ou de dimension 3 ou plus... Les cases de cette table permettent alors de stocker les résultats des sous-problèmes.

Les étapes suivies en programmation dynamique peuvent être résumées ainsi :

1. **Formule de récursivité :**
   Obtention de la formule de récursivité liant la solution d'un problème à celle de sous-problèmes.

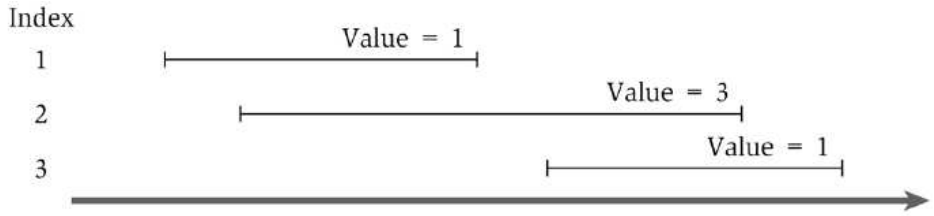
2. **Définition et Initialisation de la table :**
   Selon le nombre de paramètres entiers qui varient, on utilise une table de dimension 1, 2 ou plus.
   L'initialisation de la table correspond aux cas de base de la formule de récursivité obtenue à l'étape 1.

3. **Remplissage de la table :**
   Cette étape consiste à résoudre les sous-problèmes de taille de plus en plus grande à l'aide de boucles, en se servant bien entendu de la formule obtenue à l'étape 1.

4. **Lecture de la solution :**
   Contrairement aux exemples traités précédemment, la solution du problème de départ n'est pas toujours stockée dans la dernière case de la table. Souvent au contraire, pour avoir cette solution il est nécessaire de déchiffrer la table obtenue. Il arrive souvent que l'on soit amené (voir l'exemple du "sac à dos") à lire la table en suivant un chemin inverse à celui effectué à l'étape 3.

Exemple 1 : Ordonnancement de tâches

Ce problème est aussi connu sous sa formulation anglosaxonne : "weigthed interval scheduling".
L'objectif est d'obtenir à partir d'un ensemble de tâches chacune définie par un intervalle de temps et une pondération, un sous-ensemble de tâches compatibles (pas de chevauchement des intervalles) de pondération totale maximale.



1. Mise en forme mathématique du problème :

   (a) On note :

   - $\{t_i\}_{i\in[\![0,n-1]\!]}$ l'ensemble des n tâches à effectuées
   - $(d_i,\ f_i)$ l'intervalle de temps associé à la tâche t_i
   - v_i la pondération associée à la tâche t_i.

   On suppose les tâches t_i initialement ordonnées selon la croissance de la variable f_i.

   (b) On cherche : $J \subset [\![0, n-1]\!]$ tel que : $\begin{cases} \sum_{i\in J} v_i \quad \text{soit maximale} \\ \forall i_k \in J = \{i_1,\ i_2,\ \ldots,\ i_p\}, \quad f_{i_k} \leq d_{i_{k+1}} \end{cases}$.

   (c) Pour $i \in [\![1, n]\!]$:

   - notons P_i la valeur maximale que l'on peut obtenir en sélectionnant des tâches compatibles parmi les tâches indéxées de 1 à i.
   - notons $p(i)$ le plus grand indice $j < i$ tel que $f_j \leq d_i$ (c'est à dire, l'indice de la dernière tâche parmi $\{t_1, \ldots,\ t_{i-1}\}$ compatible avec t_i).
     Si aucune de ces tâches est compatible avec t_i alors on pose $p(i) = 0$.

   (d) Formule de récursivité :

   La valeur de P_i s'obtient de deux façons différentes selon que la tâche t_i est retenue ou non :

   - Si la tâche t_i n'est pas retenue, alors $P_i = P_{i-1}$.
   - Si la tâche t_i est retenue, alors les autres tâches retenues doivent être compatibles avec t_i. Il faut donc les choisir parmi les tâches d'index 1 à $p(i)$.
     On obtient alors la valeur maximale P_i et ajoutant v_i à la valeur maximale $P_{p(i)}$: $P_i = v_i + P_{p(i)}$.

   Nous avons ainsi la formule de récursivité suivante :

$$P_i = \max(P_{i-1},\ v_i + P_{p(i)})$$

(e) Tableau de stockage :

La valeur P_i ne dépendant que d'un seul indice entier i, nous utiliserons un tableau ligne à n composantes. La première composante de ce tableau est initialisée à $T[0] = 0$ et $T[1] = v_1$ (afin que l'indice du tableau corresponde à l'indice des tâches).

Les valeurs $p(i)$ pourront s'obtenir à l'aide d'une fonction locale p.

(f) Algorithme :

On a supposé ici que les n tâches sont ordonnées selon la croissance des temps de fin.
On considère un vecteur v de $n+1$ triplets $(d_i,\ f_i,\ v_i)$ pour coder l'ensemble des tâches à traiter.
Pour assurer la cohérence des indices des différents tableaux avec les numéros des tâches traitées, nous pouvons choisir $d_0 = f_0 = v_0 = 0$.

CAML
```
let schedule v = let n = vect_length v in
                 let T = make_vect n 0 and
                     P = make_vect n (-1) in    (* Stockage des valeurs pi*)

            (* on commence par programmer la fonction p *)

                 let p u i = let k = ref (-1) and
                                 (di,fi,vi) = u.(i) in
                             for j = 0 to (i-1) do let (dj,fj,vj) = u.(j) in
                                                      if fj <= di then k := j;
                                                   done;
                             !k in

            (* on programme alors le calcul de T.(i) *)

                 let (d0,f0,v0) = v.(0) in
                 T.(0) <- v0;
                 for i = 1 to (n-1) do let (di,fi,vi) = v.(i) in
                                       P.(i) <- p v i;
                                       T.(i) <- max T.(i-1) (T.(P.(i)) + vi)
                                    done;
T,P;;   (* On renvoie la liste des valeurs MAX et celle des indice compatibles *)

schedule [|(0,0,0);(1,4,3);(3,6,5)|];;
```

Question : Comment maintenant procéder pour identifier les tâches retenues ?

Justifier la validité du programme suivant :

CAML
```
let T,P = schedule [|(0,0,0);(2,5,3);(3,6,2);(5,7,2);(2,8,1);(6,9,3);(5,10,5);(5,12,4)|];;

let decode T P = let L = ref [] and      (* Pour stocker les indices des tâches sélectionnées*)
                     n = vect_length T in
                 let k = ref (n-1) in    (* Indice de la tâche potentiellement sélectionnée *)
                 while !k <> (-1) do
                       if !k = 0 then L := 0::(!L)
                                 else if T.(!k) = T.(!k-1) then k := !k - 1
                                                           else begin L := (!k)::(!L);
                                                                      k := P.(!k)
                                                                end;
                                 done;
                 !L;;

decode T P;;
```

Exemple 2 : Problème du sac à dos

Soit un ensemble de n objets $E_n = [\![1, n]\!]$, et un sac à dos pouvant contenir une masse maximale W. Chaque objet a une masse $w_i \in \mathbb{N}^*$ et une valeur $v_i \in \mathbb{R}^{+*}$. Le problème consiste à choisir un ensemble d'objets parmi les n objets (un objet ne pouvant être choisi qu'une seule fois) de telle manière que la valeur totale soit maximisée, sans dépasser la capacité W du sac.

En termes mathématiques, le problème se formule ainsi :

$$\text{On cherche } (x_1,\ x_2,\ \ldots,\ x_n) \in \{0,\ 1\}^n \text{ tel que : } \begin{cases} \displaystyle\sum_{k=1}^{n} x_i v_i \text{ soit maximal} \\ \displaystyle\sum_{k=1}^{n} x_i w_i \leq W \end{cases}$$

Idée 1 :

Un simple algorithme pour ce problème est de tester toutes les possibilités en mémorisant celle qui répond temporairement au problème. Cependant, on montre facilement que la complexité de cet algorithme est un $O(2^n)$.

Idée 2 : en cherchant une formule de récursivité

Notons : P_i^j le gain maximum généré par le choix $\begin{cases} \text{parmi les i premiers objets} \\ \text{d'objets dont la somme des poids ne dépasse pas } j \end{cases}$.

Résoudre le problème revient alors à trouver la valeur de P_n^W.

1. Propriété récursive du problème :

   Pour calculer P_i^j la séquence d'objets peut être divisée en deux sous-ensembles : les $(i-1)$ premiers objets et l'objet i. L'objet i est alors soit choisi soit ignoré dans le calcul de P_i^j.

   — On commence par tester si le poids de l'objet i ne dépasse pas la capacité j que l'on s'est imposée.
   Si c'est le cas, alors nous avons $P_i^j = P_{i-1}^j$.

   — si la masse de l'objet i est inférieure à j, alors nous devons considérer deux possibilités :
   - si l'on choisi l'objet i alors il contribue à la solution optimale et nous avons :

   $$P_i^j = P_{i-1}^{j-w_i} + v_i$$

   - Si l'objet i n'est pas choisi dans la solution optimale alors la capacité du sac est inchangée.
   La solution optimale est donc celle obtenue en ne considérant que le $i-1$ premiers objets, soit :

   $$P_i^j = P_{i-1}^j$$

   Pour trouver P_i^j il suffit alors de prendre le maximum entre le cas où l'objet est choisi ou ignoré :

   $$\boxed{P_i^j = \max(P_{i-1}^{j-w_i} + v_i,\ P_{i-1}^j)}$$

2. Définition et Initialisation du tableau :

   La valeur P_i^j dépendant des deux paramètres entiers i et j, on utilise une matrice pour stocker les valeurs des sous problèmes.

   Les cas de base sont : $P_i^j = 0$ pour $i = 0$ ou $j = 0$.

3. Remplissage de la table :

Cela nous amène aux relations récursives suivantes :

$$P_i^j = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ P_{i-1}^j & \text{si } w_i > j \text{ et } i > 0 \\ \max(P_{i-1}^{j-w_i} + v_i,\ P_{i-1}^j) & \text{sinon} \end{cases}$$

En CAML, l'algorithme de programmation dynamique se programme alors ainsi :

CAML

```
(* O : le vecteur contenant les objets susceptibles d'être emportés *)
(* W : la masse maximale admissible par le sac à dos               *)

let sac_a_dos O W = let n = vect_length O in
     let P = make_matrix (n+1) (W+1) 0 in
     for i = 1 to n do
         for j = 1 to W do
             let (vi,wi) = O.(i-1) in
             if wi > j then P.(i).(j) <- P.(i-1).(j)
                       else P.(i).(j) <- max (P.(i-1).(j)) (P.(i-1).(j-wi)+vi)
                        done;
                   done;
P.(n).(W);;

sac_a_dos [|(1,1);(6,2);(18,5);(22,6);(28,7)|] 11;;
```

Nous avons ainsi rempli une table de valeurs dont la dernière case d'indice $(n,\ W)$ (n étant le nombre d'objets que nous souhaitons emporter et W la masse totale admissible) donne la valeur de l'ensemble de ces objets. Cependant, pour répondre à la question "Quels sont les objets que nous allons emporter avec nous ?", il nous reste encore à procéder à un déchiffrement de cette table.

4. Complexité :

Il est clair que cette complexité est dominée par les deux boucles " for " imbriquées l'une dans l'autre : une est itérée W fois et l'autre est itérée n fois. Par conséquent, la complexité de tout l'algorithme est un $O(Wn)$.

5. Exemple :

Soient les données suivantes relatives aux objets. La capacité maximal du sac est 11 :

item	valeur	poids
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

En appliquant l'algorithme ci-dessus, on obtient la table P suivante :

(i, j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	40

La solution optimale est donc donnée par $P[5,\ 11] = 40$.

Nous obtenons donc la valeur maximale que peut contenir le sac, mais le résultat ne nous dit pas quels sont les objets qui ont été sélectionnés. Pour retrouver les objets faisant partie de la solution optimale, on procède comme suit :

- $P[5,11]$ est donné par $P[4,11]$, car lors du calcul de $P[5,11] = \max(P[4,11],\ P[4,11-w_5]+v_5) = P[4,11]$. Cela signifie que l'objet 5 ne fait pas partie de la solution optimale.
- Ensuite $P[4,11]$ est donné par $P[3,11-w_4]+v_4 = P[3,5]+v_4$.
  Autrement dit, l'objet 4 est choisi et les autres objets choisis sont parmi les 3 premiers objets.
- Ensuite $P[3,5]$ est donné par $P[2,5-w_3]+v_3 = P[2,0]+18$.
  L'objet 3 est choisi et les autres objets choisis sont parmi les 2 premiers objets.
- En continuant de la sorte, on trouve que $P[2,0] = P[1,0] = P[0,0]$, ce qui signifie qu'aucun autre objet n'a été choisi.

Les objets faisant partie de la solution optimale sont donc : l'objet 4 et l'objet 3

Voici le programme de décodage du tableau obtenu à l'étape précédente :

CAML
```
(* T : le tableau obtenu à l'étape précédente                        *)
(* O : la liste des objets susceptibles d'être emportés dans le sac *)

let decode T O = let N = (vect_length T - 1)
                 and W = (vect_length T.(0)) -1
                 in let rec aux T O = function
                     | (0,j) -> []
                     | (i,0) -> []
                     | (i,j) -> let (v,m) = O.(i-1) in
                                match T.(i).(j) = T.(i-1).(j)
                                with
                                | true -> aux T O (i-1,j)
                                | false -> i::(aux T O (i-1,j-m))
                in aux T O (N,W);;
```

Exemple 3 : Calcul de la distance d'édition

Un exemple classique d'utilisation de la *distance d'édition* est lorsque Google renvoie le même résultat de recherche lorsqu'on lui entre "dynamic" et "dymanic" comme mots clés.

Le problème que l'on cherche à résoudre est de mesurer la similitude entre deux chaînes de caractères.

On définit sur les mots trois opérations élémentaires :

- la substitution : on remplace une lettre par une autre,
- l'insertion : on ajoute une nouvelle lettre,
- la suppression : on supprime une lettre.

Par exemple, sur le mot `carie`, si on substitue c en d, a en u et si on insère t après le i, on obtient `durite`.
La **distance d'édition** entre deux mots U et V est le nombre minimal d'opérations pour passer de U à V.

Ainsi on peut démontrer que :
— la distance de `carie` à `durite` est 3 : deux substitutions et une insertion.
— La distance de `aluminium` à `albumine` est 4 : une insertion b, une substitution i en e et 2 suppressions u et m.

En notant U = ua et V = vb avec :

— ε un mot vide,
— v et u deux mots quelconques,
— $|v|$ le nombre de caractères d'un mot v
— a et b deux caractères distincts,

Formule de récursivité :

Il y a plusieurs façons de transformer U en V :

- Dans le cas où $a = b$, il suffit de tranformer u en v.
- Sinon :

M1 : On commence par transformer u et v, puis on substitue a en b
M2 : On commence par transformer u en V et on élimine le a
M3 : On commence par transformer U en v et on ajoute le b

Ces différentes éventualités nous donnent alors la formule de récursivité suivante :

1. $d(ua,\ va) = d(u,\ v)$
2. $d(ua,\ vb) = 1 + \min(d(u,\ v),\ d(ua,\ v),\ d(u; vb))$

Cas de base :

1. $d(\varepsilon,\ v) = |v|$ (on fait $|v|$ insertions à partir de ε)
2. $d(u,\ \varepsilon) = |u|$ (on fait $|u|$ suppressions dans u)

Travail à effectuer :

Nous allons concevoir un programme dynamique utilisant la formule de récursivité et les cas de base précédents. Notons U et V les deux mots traités avec n le nombre de lettre de U et m le nombre de lettres de V.

On décide de stocker les résultats intermédiaires dans une matrice T de taille $(n+1) \times (m+1)$, telle que $T(i, j)$ soit la distance d'édition du mot constitué des i premières lettres de U au mot constitué des j premières lettres de V. La première colonne et la première ligne correspondant au cas où l'un des mots est vide. La valeur de $T(n, m)$ correspond alors à la distance d'édition de U à V.

1. Ecrire en programmation dynamique une fonction `Distance` qui calcule la distance de deux chaînes de caractères.

   Ce programme appliqué aux mots `carie` et `durite` permet ainsi de construire le tableau suivant :

			c	a	r	i	e
		0	1	2	3	4	5
	0	0	1	2	3	4	5
d	1	1	1	2	3	4	5
u	2	2	2	2	3	4	5
r	3	3	3	3	2	3	4
i	4	4	4	4	3	2	3
t	5	5	5	5	4	3	3
e	6	6	6	6	5	4	3

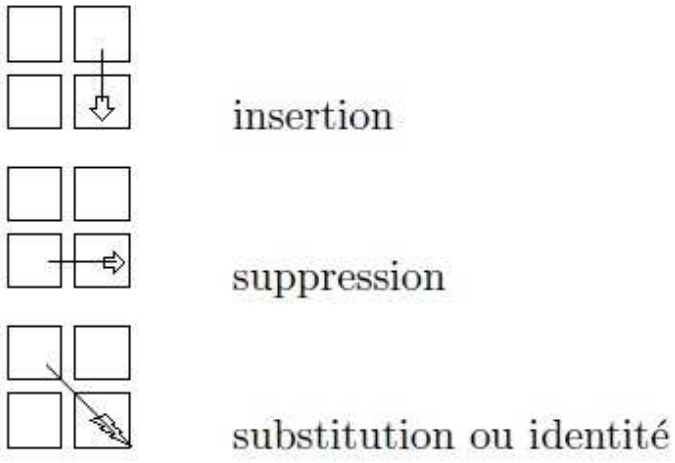
2. Décodage de la matrice :

   On veut maintenant connaitre la suite d'opérations qui mène de U à V.
   Pour cela, on peut visualiser les transformations par un petit schéma :

```
c a r i - e        a l - u m i n i u m
    | | |        | |   | | | |
d u r i t e        a l b u m i n e - -
```

Deux lettres identiques sont signalées par |. Un espace dans le mot de départ correspond à une insertion, un espace dans le mot d'arrivée correspond à une suppression, et une substitution est représentée par les deux lettres face à face, sans |.

Il est possible de connaître la dernière opération appliquée en regardant comment la valeur de $T(u_{max},\ v_{max})$ a été obtenue. Par exemple, si $T(u_{max},\ v_{max})$ est égal à $T(u_{max-1},\ v_{max}) + 1$, alors la dernière opération est une insertion de $V(v_{max})$.

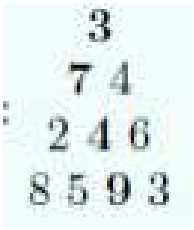


Retrouvez à la main à partir de la table de l'exemple la suite des transformations pour passer de carie à durite.

Ecrire une procédure qui prend pour arguments deux chaînes de caractères et affiche la suite d'opérations sous la forme décrite ci-dessus. Pour cela, vous devez utiliser la matrice T et construire l'historique des transformations en remontant dans la table à partir de $T(u_{max},\ v_{max})$.

3 Chemin à valeur maximale dans un triangle isocèle

Déterminer un algorithme dynamique permettant de déterminer le chemin allant du sommet à la base tel que la somme des valeurs par lequel il passe soit maximale.



Ce triangle pourra être codé dans un vecteur de vecteurs dont chaque composante correspond à une ligne du triangle.

4 Mémoïzation

Pour optimiser le temps de réalisation d'un algorithme, il est intéressant de garder en mémoire les résultats obtenus lors de précédentes utilisations (expl : fonction *remember* de Maple pour les fonction récursives). C'est ce que fait la *mémoire cache* pour la navigation sur le web.

Dans le cas de la programmation dynamique d'une fonction f, nous pouvons envisager de stocker systématiquement toutes les valeurs calculées dans un tableau mémoire T défini antérieurement de façon globale, initialisé avec les cas de base et dont la taille a été fixée pour pouvoir contenir toutes les combinaisons d'arguments envisageables en pratique. L'amélioration de la fonction f se programme alors de la façon suivante :

```
let f x1 ... xn = if T[x1,...,xn] <> valeur d'initialisation du tableau T
                  then renvoyer T[x1,...,xn]
                  else let s = programme calculant f(x1,...,xn) in
                       T[x1,...,xn] <- s:
                       renvoyer s;;
```

Ce choix de mémoïzation engendre la mobilisation d'un espace mémoire supplémentaire, mais permet d'éviter de refaire des calculs d'ores et déjà effectués lors d'une utilisation antérieure de la fonction. Si on y perd en complexité spatiale, on y gagne cependant beaucoup en complexité temporelle.

Bien entendu, ce principe de *mémoïzation* s'applique à toutes les formes de programme : itératifs et récursif.