CHAPITRE 5

PROGRAMMATION DYNAMIQUE

Sommaire

En informatique, la programmation dynamique est une méthode algorithmique pour résoudre des problèmes d'optimisation. La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.

I Définition, programmation dynamique versus méthode "diviser pour mieux régner"

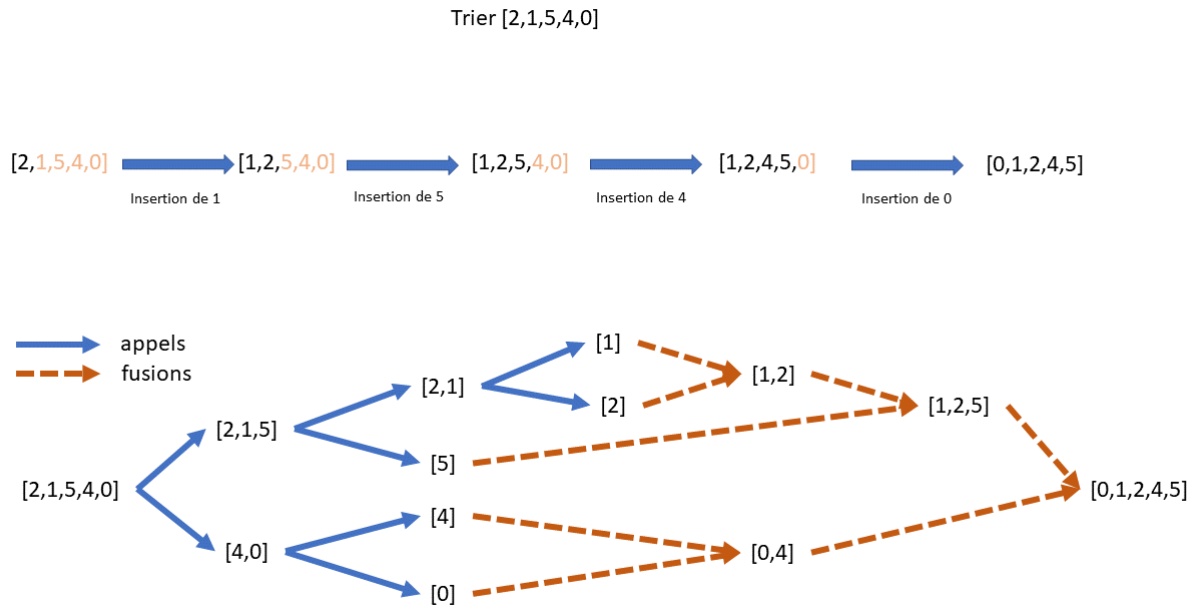
Le principe de la programmation dynamique est de décomposer le problème en sous-problème "de taille plus petite" dans un sens qu'il faut préciser selon le problème. La méthode "diviser pour mieux régner" consiste à diviser le problème en sous-problème indépendant pour ensuite résoudre le problème original de manière récursive.

Exemple du tri d'une liste

On peut voir la méthode de *tri par insertion* comme une méthode de programmation dynamique, le *tri fusion* est, quant à lui, utilise la méthode "diviser pour mieux régner".
Le tri par insertion résout les problèmes :

- je tri le premier élément ;
- je tri les deux premiers en insérant le second
- je tri les trois premiers en insérant le troisième
- etc...
- je tri la liste en insérant le dernier

A contrario, le tri fusion consiste à couper la liste en deux parts égales (ou presque) et à demander de les trier par appel récursif. Lorsque deux listes sont triées, on les fusionne avec une méthode linéaire.



EN HAUT LE TRI PAR INSERTION ET EN BAS LE TRI FUSION.

II Programmation ascendante, descendante

Il existe deux manières de programmer par la méthode de programmation dynamique :

Programmation ascendante

On utilise une méthode itérative pour passer des problèmes les plus petits aux problèmes les plus gros.

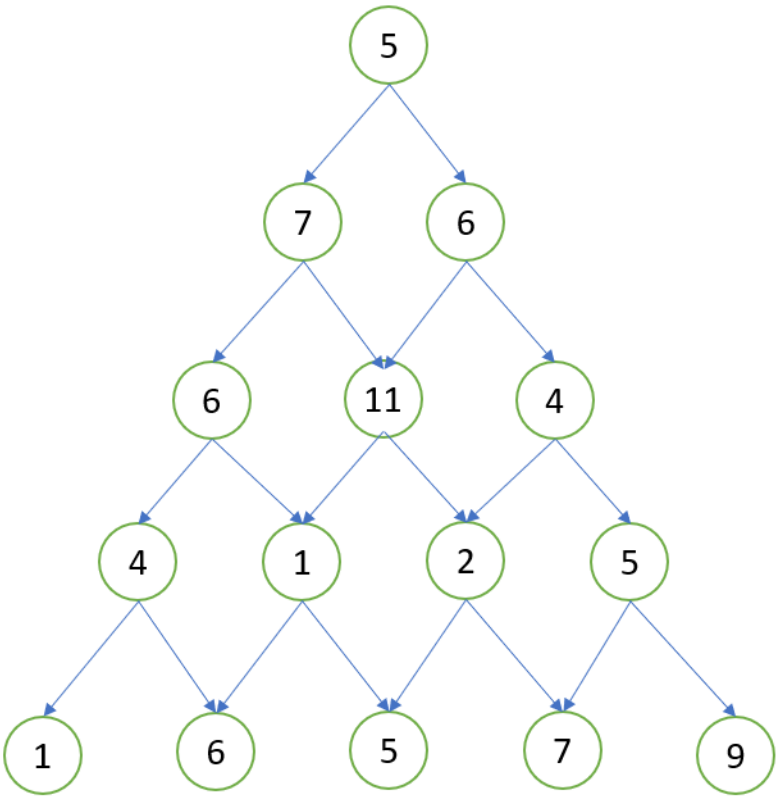
Programmation descendante

On utilise une méthode récursive pour résoudre les problèmes les plus gros à l'aide des problèmes les plus petits. Lorsque le graphe de dépendance des différents problèmes n'est pas un arbre (c'est-à-dire lorsqu'on a besoin de faire plusieurs fois le même appel récursif), on mémorise les résultats des appels récursifs : c'est la **mémoïsation**.

III Exemples

III.1 La pyramide de nombres

Dans une pyramide de nombres, on cherche, en partant du sommet de la pyramide, et en se dirigeant vers le bas à chaque étape, à maximiser le total des nombres traversés :

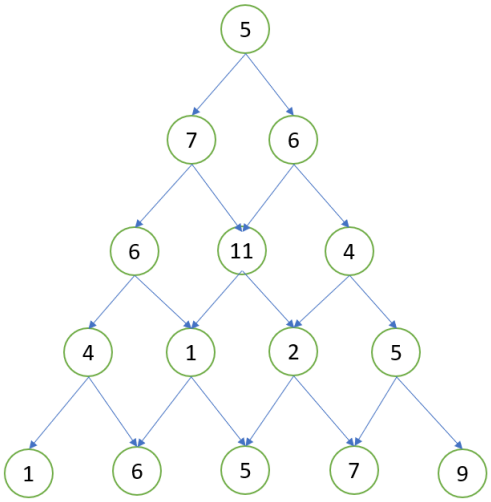


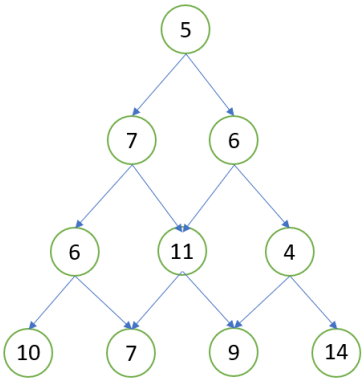
Une idée naïve consisterait à faire toutes les trajectoires possibles et de calculer le total pour chaque trajectoire, or, si il y a n niveau dans la pyramide, le nombre de trajectoire possible est de 2^{n-1} ce qui donne un algorithme de complexité exponentielle en temps.

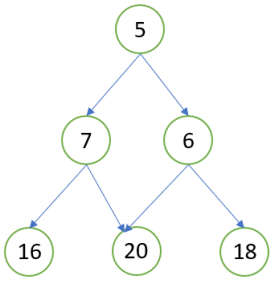
L'utilisation de la programmation dynamique permet d'obtenir un algorithme efficace en définissant des sous-problèmes, en écrivant une relation de récurrence, puis en donnant un algorithme (avec méthode ascendante ou descendante) :

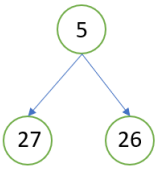
Pour toute position x dans la pyramide, notons $v(x)$ le nombre écrit à cette position et $c(x)$ le total des nombres traversés dans un chemin dont le total est maximal et partant de x. Les sous-problèmes consistent à calculer les valeurs de $c(x)$ pour tout x. Le problème initial consiste à calculer $c(x)$ lorsque x est le sommet de la pyramide. Pour cela, on va calculer les valeurs $c(x)$ niveau par niveau en partant du rez-de-chaussée :

$c(x) = v(x)$ pour toute position x situé au rez-de-chaussée de la pyramide $c(x) = v(x) + \max(c(g(x)), c(d(x)))$ pour toute autre position x, où $g(x)$ et $d(x)$ sont les positions inférieurs gauche et droite sous la position x.

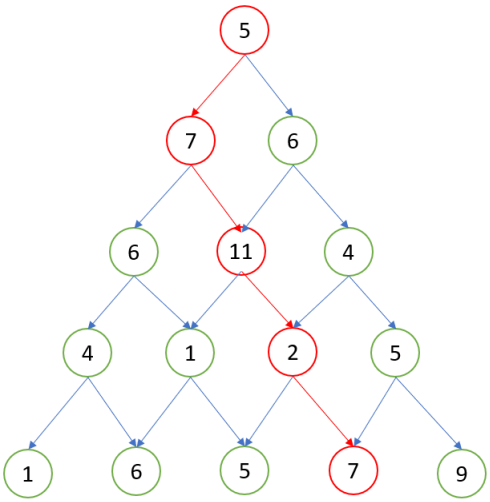








meilleur chemin :



Si on cherche à calculer directement par la définition récursive, on évalue plusieurs fois la même valeur par exemple, le meilleur chemin pour arriver à 11 de l'étage 2 est calculé pour 7 et 6 de l'étage 3. Il faut donc faire de la mémoïsation pour éviter de calculer plusieurs fois la même valeur. En ascendant, on stocke les valeurs dans, par exemple, une liste de listes, en partant du rez-de-chaussée au sommet.

III.2 Produit de matrice et parenthèsage optimal

Nous savons que la multiplication matricielle est associative c'est à dire que, lorsque cela a un sens, les produits $A \times (B \times C)$ et $(A \times B) \times C$ sont égaux ce qui fait qu'on note $A \times B \times C$ ou ABC sans préciser la place des parenthèses lorsqu'on ne s'intéresse qu'au produit lui-même. Par contre, les choses changent quand il s'agit de réaliser le calcul. Lorsque $A \in \mathcal{M}_{p,q}$, $B \in \mathcal{M}_{q,r}$, $C \in \mathcal{M}_{r,s}$, on a toujours $A(BC) = (AB)C \in \mathcal{M}_{p,s}$ mais le premier calcul aura coûté $pqs + qrs = qs(p + r)$ multiplications (de réels, complexes ou flottants) alors que le second en aura coûté $pqr + prs = pr(q + s)$.

$$\begin{pmatrix} * & * \\ * & * \\ * & * \\ * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix}$$

Dans cet exemple, la première méthode donne $qs(p+r) = 72$ alors que $pr(q+s) = 120$. Il est donc intéressant de planifier l'ordre des calculs, pour effectuer le moins d'opérations possibles. On peut montrer que le nombre de parenthèsages possibles pour n matrices est :

$$c_n = \sum_{i=0}^{n-1} c_i \times c_{n-1-i} = \frac{1}{n+1} \begin{pmatrix} 2n \\ n \end{pmatrix} \underset{n\to+\infty}{\sim} \frac{4^n}{\sqrt{\pi} n^{3/2}}$$

Ce sont les nombres de **Catalan** ($c_0 = 1$).

Considérons donc une suite de n matrices $(A_0, A_1, \ldots, A_{n-1})$ telle que les produits $A_i A_{i+1}$ soient définis. Pour $0 \leq i \leq n-1$, on note ℓ_i le nombres de lignes de A_i et on pose $\ell_n = c_{n-1}$, nombre de colonnes de A_{n-1} ; on a donc $A_i \in \mathcal{M}_{\ell_i,\ell_{i+1}}$. Nous voulons savoir quels parenthésages permettront d'optimiser le nombre de multiplications pour calculer le produit des A_i.

Imaginons qu'un tel parenthésage optimal conduise à effectuer comme dernière opération le produit de deux matrices $(A_0 \ldots A_k) \times (A_{k+1} \ldots A_{n-1})$. Ce dernier produit matriciel demandera à lui seul $\ell_0 \times \ell_{k+1} \times \ell_n$ multiplications. Le nombre total des multiplications sera donc égal à la somme de trois termes :
- nombre de multiplications pour le calcul de $(A_0 \ldots A_k)$;
- nombre de multiplications pour le calcul de $(A_{k+1} \ldots A_{n-1})$;
$-\ell_0 \times \ell_{k+1} \times \ell_n$.

Si cette somme de trois termes est optimale, le premier et le second terme sont aussi obtenus avec des parenthésages optimaux. En effet, si le parenthésage pour calculer $(A_0 \ldots A_k)$ n'est pas optimal, on peut le remplacer par un meilleur choix ce qui n'a pas d'incidence sur les façons de calculer $(A_{k+1} \ldots A_{n-1})$ ou le dernier produit. On améliore ainsi le score global ce qui contredit le fait que notre parenthésage pour $A_0 \ldots A_{n-1}$ est optimal.

Cette propriété est fondamentale, elle va nous permettre d'élaborer une stratégie pour déterminer les parenthésages optimaux. Notons $m(i,j)$ le nombre minimal de multiplications pour calculer $A_i \times \ldots \times A_j$ lorsque $0 \leq i < j \leq n-1$, et posons $m(i,i) = 0$. Pour calculer le produit $A_i \ldots Aj$ on peut placer une) après A_k pour $i \leq k < j$ et calculer séparément les deux produits $(A_i \ldots A_k)$ et $(A_{k+1}..A_j)$. Ce choix étant fait, le nombre minimal de multiplications est :

$$m(i,k) + m(k+1,j) + \ell_i \ell_k \ell_{j+1}$$

$m(i,j)$ est donc la valeur minimale parmi les $m(i,k) + m(k+1,j) + \ell_i \ell_k \ell_{j+1}$.

Il ne reste plus qu'à résoudre à l'aide de la programmation dynamique :

- soit par une méthode de programmation ascendante : on calcule les $m(i,j)$ lorsque $i = j$, puis $j = i+1$, $j = i+2$ et ainsi de suite ce qui revient à remplir la partie supérieure d'un tableau en partant de la diagonale.
- soit par une méthode de programmation descendante : on calcule les $m(i,j)$ en appelant récursivement la valeur de $m(i,k) + m(k+1,j) + \ell_i \ell_k \ell_{j+1}$ avec $k \in [\![i, j-1]\!]$. L'algorithme se

termine car $m(i,j)$ fait appel à des $m(u,v)$ tel que $|u-v| < |i-j|$. Le graphe de dépendance n'est clairement pas un arbre, il faut faire de la mémoïsation.

Il faudra stocker (dans les deux cas), le parenthèsage optimal en conservant quel élément donne le minimum.

Ces problèmes ont une chose en commun : la solution optimale pour un problème de taille n (par exemple, choisir un parenthésage optimal pour un produit de n matrices) se construit à partir de solutions également optimales pour des problèmes de tailles inférieures. On dit que les sous-problèmes sont indépendants : modifier la solution de l'un d'eux n'impose pas de modifier les autres pour conserver la solution globale. On remarquera que tous les problèmes d'optimisation ne présentent pas cette propriété.

IV Problèmes éligibles à la programmation dynamique

Comme on l'a vu dans les exemples, pour qu'un problème d'optimisation soit éligible il faut que la solution de l'optimisation d'un problème de taille n s'obtienne avec la ou les solutions solutions de l'optimisation d'un problème de taille inférieur ou égale à $n-1$.
C'est bien le cas, par exemple, dans le problème de parenthèsage des matrices, où la taille correspond aux nombres de matrices.

Théorème 5.1: Principe d'optimalité de Bellman

une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous-problèmes.

Il faut remarquer que ce n'est pas toujours le cas, prenons comme exemple simple celui de la recherche de chemins optimaux. On considère un graphe orienté $G = (S, A)$ et on s'intéresse à :
- la recherche d'un chemin avec un minimum d'arcs reliant un sommet u à v ;
- la recherche d'un chemin sans boucle avec un maximum d'arcs reliant u à v.

Exercice 5.2: Expliquer pourquoi, dans le premier cas, un chemin optimal qui relie u à v en passant par x est tel que les chemins $u \to x$ et $x \to v$ sont optimaux

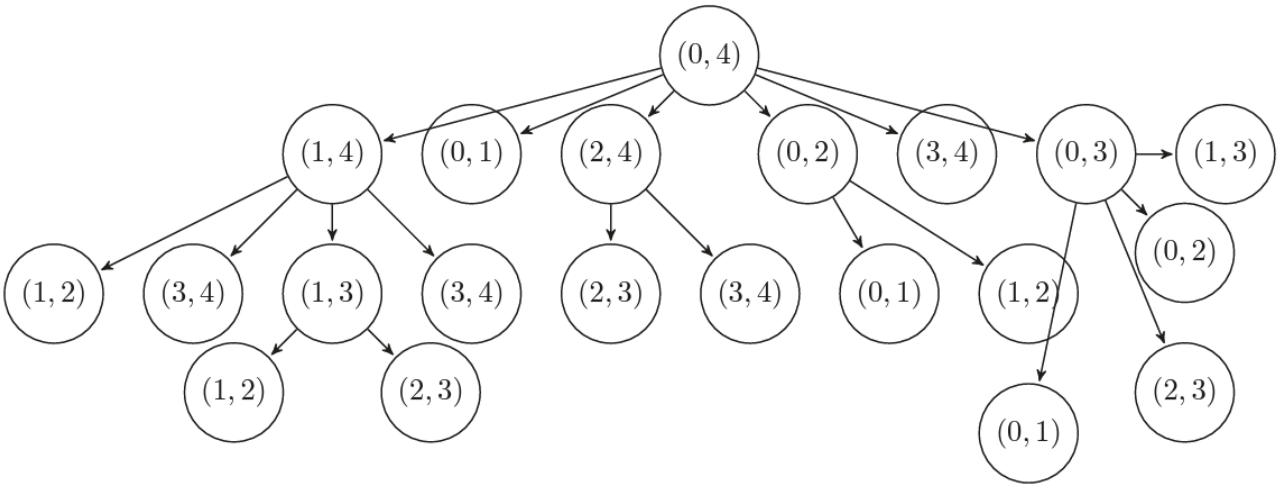
E

E

Exercice 5.3: Donner un exemple dans le second cas qui contredit la propriété de l'exercice précédent.

E

Dans la programmation dynamique, les sous-problèmes n'ont pas besoin d'être indépendant, il faut juste stocker les données pour éviter de recalculer les mêmes appels, par exemple, dans le cas $n = 5$ d'un produit de matrices, on obtient l'arbre d'appel suivant :



Il y a clairement des appels effectués plusieurs fois (et n est petit), il faut faire de la mémoïsation.

Dans le cas d'un algorithme "diviser pour mieux régner, les sous-problèmes d'un même niveau d'appel doivent être indépendants, c'est le cas dans l'arbre d'appel pour trier la liste [2,1,5,4,0] par tri fusion (cela serait aussi le cas avec quick-sort qui demanderait de trier [1,0] et [5,4] avec comme pivot 2, le premier élément.).