Programmation dynamique

1. Principe général

Quelquefois, en essayant de résoudre un problème de taille n par l'approche descendante (Diviser pour Résoudre), on s'aperçoit que la décomposition génère plusieurs sous problèmes identiques. Si on résout chaque sous exemplaire séparément sans tenir compte de cette duplication on obtient un algorithme très inefficace. Par contre, si on prend la précaution de résoudre chaque sous exemplaire différent une seule fois (en sauvegardant, par exemple, les résultats déjà calculés) on obtient un algorithme performant.

Idée de base :
Eviter de calculer deux fois la même chose, normalement en utilisant une table de résultats déjà calculés, remplie au fur et à mesure qu'on résout les sous problèmes.

Remarques
C'est une méthode ascendante : On commence d'habitude par les sous problèmes les plus petits et on remonte vers les sous problèmes de plus en plus difficiles.
La programmation dynamique est souvent employée pour résoudre des problèmes d'optimisation satisfaisant le principe d'optimalité: "Dans une séquence optimale (de décisions ou de choix), chaque sous-séquence doit aussi être optimale". Un exemple de ce type de problème est le plus court chemin entre deux sommets d'un graphe.

2. Applications

Exemple 1: Calcul de C_n^p

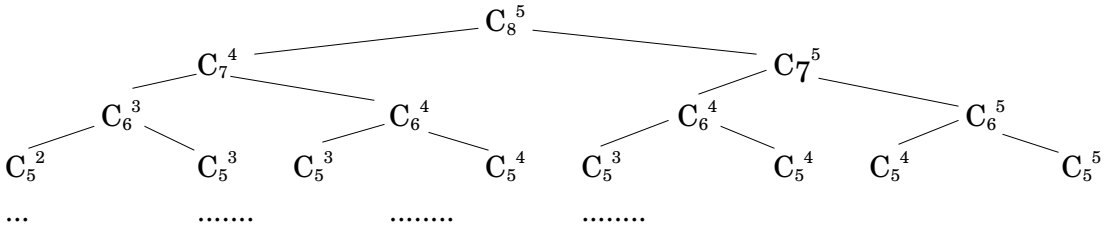
Il existe une décomposition connue pour le calcul de C_n^p :

$C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$
$C_0^0 = C_n^0 = C_n^n = 1$

a) On peut donner un algorithme récursif (c'est l'approche descendante : Diviser pour Résoudre) utilisant la définition de C_n^p.

```
Fonction C_rec (n,p)
Si (n=p) ou (p=0):    retourner(1)
Sinon                 retourner(C_rec(n-1,p-1) + C_rec(n-1,p))
Fsi
```

Mais cette décomposition va générer un nombre important de sous problèmes identiques. Par exemple pour calculer C_8^5 l'arbre des appels récursifs est :



On remarque qu'il existe beaucoup de sous exemplaires qui se répètent. Ceci influence négativement la complexité de l'algorithme qui est en (C_n^p).

b) On peut aussi faire le calcul par programmation dynamique.(triangle de Pascal)

	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

↓ (remplissage de la table)

Le calcul se fait ligne par ligne et de gauche à droite, c-a-d que les sous exemplaires les plus simples sont calculés en premiers et les résultats, que l'on sauvegarde dans la table, servent à calculer les exemplaires plus grands.

```
Fonction C_Dyn (n,p)
var
        Mat[0..n,0..p]
Pour i=0,n :
        Pour j=0,Min(i,p) :
                Si i=j ou j=0 :
                        Mat[i,j] := 1
                Sinon
                        Mat[i,j] := Mat[i-1,j-1] + Mat[i,j]
                Fsi
        Fp
Fp
retourner(Mat[n,p])
```

Cet algorithme est en O(np).

Exemple 2: La suite de fibonacci

La définition de cette suite est donnée par la décomposition :
$f_0 = 1$, $f_1 = 1$ et
$f_n = f_{n-1} + f_{n-2}$ pour $n > 1$

L'approche descendante (Diviser pour Résoudre) donne l'algorithme suivant:

```
Fib_Rec(n:Entier) : Entier;
Si n<=1 :       Fib_Rec := 1
Sinon           Fib_Rec := Fib_Rec(n-1) + Fib_Rec(n-2)
Fsi
```

Comme dans l'exemple précédent, cet algorithme a un coût exponentiel car l'arbre des appels récursifs généré contient beaucoup d'appels identiques.

L'approche par Programmation Dynamique suggère de commencer par les sous-problèmes les plus petits (Fib(0) et Fib(1)) est de calculer les sous-problèmes plus complexes (Fib(2), Fib(3), Fib(4), ... Fib(n)) en gardant la trace des résultats intermédiaires nécessaires au calcul.
Dans ce problème, les résultats intermédiaires nécessaires au calcul d'un sous-problème donné (Fib(i)) sont les deux sous-problèmes précédents (Fib(i-1) et Fib(i-2) car Fib(i)=Fib(i-1) + Fib(i-2)), d'où l'algorithme suivant:

```
Fib-Dyn(n:Entier) : Entier;
var
        i,j,k : Entier;
j := 0;  i := 1;
Pour k=1,n :
        j := i+j;
        i := j-i
Fp;
Fib_Dyn := j
```

Cet algorithme est en o(n).

Exemple 3: Série mondiale

A et B disputent une série de matchs (au maximum 2n-1).
L'équipe gagnante est celle qui remporte n victoires.
La probabilité que A remporte une victoire, lors d'un match, est : q1
La probabilité que B remporte une victoire est : q2 = 1-q1
Il n'y a pas de match nul.

On définit P(i, j) comme la probabilité que A remporte la série, sachant qu'elle doit encore gagner 'i' victoires alors qu'il reste 'j' victoires à B pour gagner.

On a ainsi :
P(0, j) = 1 (A a déjà gagné la série et j > 0)
P(i, 0) = 0 (A a déjà perdu la série et i > 0)
P(0, 0) indéfini
P(i, j) = q1*P(i-1, j) + q2*P(i, j-1) i, j > 0

Algorithme récursif :

```
P(i, j) :
Si i = 0 et j > 0 :
        P := 1
Sinon
        Si i > 0 et j = 0 :
                P := 0
        Sinon
                Si i > 0 et j > 0 :
                        P := q1*P(i-1,j) + q2*P(i, j-1)
                Fsi
        Fsi
Fsi
```

T(k) = temps d'exécution de P(i, j) avec k = i+j

Equation de récurrence : $T(k) = 2T(k-1) + b$

Solution : $T(k) = C_1 2^k + C_2$

donc T(k) est en $O(2^k)$ ou encore en $O(2^{i+j})$

P(n, n) prend un temps dans l'ordre de $O(4^n)$ (car $2^{2n} = 4^n$)

Programmation dynamique

Utilisation d'une table remplie diagonale par diagonale:

	0	1	2	3	4
0	1	1	1	1	1
1	0	1/2	3/4	7/8	15/16
2	0	1/4	1/2	11/16	13/16
3	0	1/8	5/16	1/2	21/32
4	0	1/16	3/16	11/32	1/2

L'algorithme pour remplir la table jusqu'à la position P[i,j] :

```
Pour S = 1, i+j
        P[0, S] := 1
        P[S, 0] := 0
        Pour k=1, S-1
                P[k, S-k] := q1P[k-1, S-k] + q2P[k, S-k-1]
        Finpour
Finpour
```

Le temps d'exécution est en $O(n^2)$ pour le calcul de P(n,n).

Exemple 4: Les plus courts chemins

Il s'agit de trouver les plus courts chemins entre chaque pair de sommets d'un graphe valué (Algorithme de Floyd).

C'est un problème d'optimisation qui vérifie le principe d'optimalité : Si le plus courts chemin (chemin optimal) entre deux sommets A et B passe par un sommet intermédiaire C, alors les portions du chemin entre A et C et entre C et B doivent forcément être optimales.

Dans un graphe formé par n sommets, l'algorithme consiste à calculer les plus courts chemins entre chaque pair de sommets en utilisant comme sommets intermédiaires, dans l'ordre et de façon successives les sommets 1, 2, 3 .. n.
A l'état initial aucun sommet intermédiaire n'est utilisé, les plus courts chemins sont donnés directement par la matrice des poids des arcs D0[1..n,1..n]. D0[i,j]=+∞ désignera l'inexistence d'un arc entre les sommets i et j.

A l'étape 1 les nouveaux chemins sont calculés par :

$$D_1[i,j] = \text{Min}\ (D_0[i,j]\ ,\ D_0[i,1] + D_0[1,j])$$

A l'étape k les nouveaux chemins sont calculés en prenant en considération tous les sommets entre 1 et k :

$$D_k[i,j] = \text{Min}\ (D_{k-1}[i,j]\ ,\ D_{k-1}[i,k] + D_{k-1}[k,j])$$

Après l'étape n, la matrice D_n contiendra les longueurs des plus courts chemins entre chaque couple de sommets (i,j).

Exemple 5: Multiplication chaînée de matrices

Sachant que la multiplication de matrices est associative :

$$M_1(M_2\ M_3) = (M_1\ M_2)\ M_3$$

et sachant que pour multiplier une matrice de p lignes et q colonnes $M_1(p,q)$ par une autre matrice de q lignes et r colonnes $M_2(q,r)$ il faut p*q*r multiplications élémentaires, le problème est de trouver le nombre minimal de multiplications élémentaires nécessaire pour multiplier une série de n matrices : $M_1\ M_2\ M_3 \ldots M_n$. Appelant ce nombre m(1,n).

Avec cette définition, m(i,j) désignera le nombre minimal de multiplications élémentaires nécessaires pour faire le produit de matrices suivant : $M_i * M_{i+1} * M_{i+2} \ldots * M_j$ et m(i,i) sera toujours égal à 0.

Supposons que les dimensions des matrices M soient stockées dans un vecteur D[0..n] de sorte que les dimensions d'une matrice M_i soient données par D[i-1] (nb de lignes) et D[i] (nb de colonnes).

On pourra alors écrire que m(i,i+1) = D[i-1]*D[i]*D[i+1]. C'est le nombre de multiplications élémentaires nécessaire au produit : $M_i\ M_{i+1}$

C'est un problème d'optimisation vérifiant le principe d'optimalité:

« Si pour faire le produit de n matrices avec un nombre minimal d'opérations élémentaires on découpe le produit des n matrices en deux sous-produits au niveau de la $i^{ième}$ matrice :
$(M_1 * M_2 * ... M_i) * (M_{i+1} * M_{i+2} * ... M_n)$
alors chacun des deux sous-produits doit aussi être optimal :
Le sous-produit P1 : $(M_1 * M_2 * ... M_i)$ doit être calculé de façon optimal (soit m(1,i) opération élémentaires), le résultat est une matrice X de dimensions : D[0] x D[i].
Le sous-produit P2 : $(M_{i+1} * M_{i+2} * ... M_n)$ doit aussi être calculé de façon optimal (soit m(i+1,n) opérations élémentaires), le résultat est une matrice Y de dimensions : D[i] x D[n].
Le produit final : X * Y nécessitera alors D[0]*D[i]*D[n] multiplications élémentaires. »

Donc si on découpe le produit initial au niveau de la $i^{ième}$ matrice, on pourra écrire :
m(1,n) = m(1,i) + m(i+1,n) + D[0]*D[i]*D[n]

En procédant par décomposition récursive pour calculer m(1,n) les sous-produits P1 et P2 on aboutira à un algorithme récursif (du type Diviser pour Résoudre) très inefficace à cause du nombre élevé de sous problèmes identiques générés par la décomposition.

L'algorithme récursif est :

```
fonction m(i,j:entier) : entier;
var
        k, x : entier;
Si i=j : retourner(0)
Sinon
        x := +
        Pour k:=i , j-1:
                x := min { x , D[i-1]*D[k]*D[j] + m(i,k) + m(k+1,j) }
        Fp
        Retourner(x)
Fsi
```

c-a-d :

$$m(1,n)=Min \begin{cases} m(1,1)+m(2,n)+D[0]*D[1]*D[n] & \text{pour: } M_1(M_2\ M_3...M_n) \\ m(1,2)+m(3,n)+D[0]*D[2]*D[n] & \text{pour: } (M_1\ M_2)(M_3...M_n) \\ ... & \\ m(1,k)+m(k+1,n)+D[0]*D[k]*D[n] & \text{pour: } (M_1...M_k)(M_{k+1}...M_n) \\ ... & \\ m(1,n-1)+m(n,n)+D[0]*D[n-1]*D[n] & \text{pour: } (M_1\ M_2...M_{n-1})M_n \end{cases}$$

En appliquant la programmation dynamique on procédera de façon ascendante pour éviter de calculer plusieurs fois le même sous-problème, en sauvegardant les résultats calculés dans une table m[1..n,1..n] qu'on remplit au fur et à mesure qu'on progresse dans la résolution.

Le remplissage de la table se fera en diagonal à l'aide de l'algorithme suivant :

```
Pour i:=1,n                          /* la 1ere diagonale */
        m[i,i] = 0                   /* car il n'y a pas de produit de matrice */
Fp;

Pour i:=1,n-1                        /* la 2e diagonale */
        m[i,i+1] = D[i-1]*D[i]*D[i+1]        /* Pour le produit : Mi * Mi+1 */
Fp;

Pour s:=2,n-1                        /* les autres diagonales */
        Pour i:=1,n-s
                m(i,i+s) =      min { m(i,k) + m(k+1,i+s) + D[i-1]*D[k]*D[i+s] }
                              i≤ k ≤ i+s-1
        Fp
Fp
Retourner( m[1,n] )
```