Chapitre II

Approches par Décomposition

II.A) Approche descendante : Diviser Pour Résoudre

II.A.1 Principe général

Diviser le problème de taille n en sous-problèmes plus petits de manière à ce que la solution de chaque sous-problème facilite la construction du problème entier. C'est une méthode descendante.

Pour résoudre un problème donné on doit d'abord résoudre les sous-problèmes engendrés par sa décomposition, puis on combine les sous-solutions obtenues pour former la solution globale.

Chaque sous-problème est résolu en appliquant le même principe de décomposition (récursivement). Ce même procédé se répète jusqu'à obtenir des sous-problèmes suffisamment simples pour être résolus de manière triviale (sans décomposition).

Le schéma général des algorithmes (récursifs) de type « diviser pour résoudre » est le suivant :

```
DPR(x) :                        // x est le problème à résoudre
  Si x est suffisamment petit ou simple :
      . Retourner  A(x)         // A(x) est un algorithme connu (peut être trivial) pour
résoudre x
  Sinon
      . Décomposer x en sous exemplaires x1,x2...xk
      . Pour i=1, k :
            yi ← DPR(xi)        // résoudre chaque sous-problème récursivement
        Fp                      //
      . Combiner les yi pour former une solution y à x
      . Retourner y
  Fsi
```

Pour pouvoir appliquer cette méthode, on doit donc :

- trouver une décomposition du problème en sous-problèmes plus simples
- avoir un algorithme A(x) capable de résoudre chaque problème simples (sans

décomposition)
- savoir comment combiner les solutions intermédiaires pour former la solution globale.

Nous allons présenter dans ce qui suit, l'application de cette méthode sur quelques exemples.

II.A.2 Applications de DPR

II.A.2.1. Tri par fusion

Une approche "Diviser Pour Résoudre" pourrait être la suivante:

Pour trier un tableau T (ou une liste) de n éléments, il faut trier deux sous-tableaux (ou sous-listes) d'environ n/2 éléments chacun (les 2 moitiés de T), puis faire l'interclassement (ou fusion) de ces deux moitiés triées. De la même façon, pour trier un tableau de n/2 éléments il faut trier deux tableaux de n/4 éléments chacun et les fusionner. On répète le même principe jusqu'à l'obtention de tableaux formé d'un seul élément (dont le tri est trivial).

Ce qui peut s'énoncer algorithmiquement, de manière récursive, comme suit:

```
Tri(T, n)
Si n = 1 :
    Retourner T                         // T est déjà trié
Sinon
    T1 ← T [1..n/2]                 // récupérer la première moitié dans T1
    T2 ← T [n/2+1..n]                   // récupérer la seconde moitié dans T2
    R1 ← Tri( T1, n/2 )             // résolution récursive
    R2 ← Tri( T2, n/2 )             // des 2 sous problèmes.
    Retourner Fusion( R1,R2 )           // combiner les 2 solutions obtenues.
Fsi
```

La fusion de 2 tableaux triés de n éléments se fait en 2n itérations (O(n)).

L'équation de récurrence associée est :

$$T(n) = \begin{cases} a & \text{si } n = 1 \\ 2T(n/2) + bn & \text{sinon} \end{cases}$$

La constante a désigne le temps nécessaire pour le test (n=1) et celui pour l'opération de retour de l'appel (retourner T). Le terme (bn) inclut le temps nécessaire pour le test, la récupération des 2 moitiés et la fusion des 2 solutions (boucle de 2n itérations).

On peut résoudre facilement cette équation de récurrence soit par substitution soit en se ramenant à une équation non homogène en procédant au changement de variable

suivant : k = $\log_2 n$ et en posant $t_k = T(2^k)$.
L'équation devient alors :
$t_k = 2t_{k-1} + b2^k$, qui est bien une équation non homogène : $t_k - 2t_{k-1} = b2^k$,
Le polynôme caractéristique étant alors : (x-2)(x-2) = 0
Donc la solution générale est de la forme $t_k = C1\ 2^k + C2\ k\ 2^k$
En revenant à n, on aura $T(n) = C1 * n + C2 * (\log_2 n) * n$

Solution : O(n*log n) ce qui est meilleur comparé à beaucoup d'algorithmes de tri en $O(n^2)$.
Voici une implémentation en langage C :

```
// T et tmp des variables globales représentant des tableaux.
int *T;      // T contiendra les éléments à trier
int *tmp;    // tmp est un tableau temporaire pour réaliser les fusions

/* Allocation et initialisation du tableau à trier * /
void init_tab(int n)  // n le nombre d'éléments du tableau
{
    int i;
    // allocation des tableaux
    T = malloc(n*sizeof(int));
    tmp = malloc(n*sizeof(int));

    // Remplissage de T avec n nombres aléatoires
    if ( T != NULL && tmp != NULL ) {
        // initialisation du générateur de nombres aléatoires
        srand((unsigned int)time((time_t *)NULL));

        // Génération de n nombre aléatoires entre 0 et 2n
        for (i=0; i<n; i++)
            T[i] = 1+(int) (2.0*n*rand()/(RAND_MAX+1.0));
    }
    else
       printf("Problème d'allocation d'espace pour les tableaux. \n");
}  // fin int_tab

/* Tri fusion (version récursive)                  * /
void tri_fus(int a, int b)
{
   int m,i,j,k;
   // cas particulier ...
   if (a>=b)
         return;

  // cas général ...
   m = (a+b)/2;
   tri_fus(a,m);      // tri de la 1ere moitié
   tri_fus(m+1,b);    // tri de la 2eme moitié

   // Fusion des 2 moitiés triées...
   i = a;
   j = m+1;
   k = 0;
```

```
   while (i<=m && j<=b)
    if (T[i] < T[j])
        tmp[k++] = T[i++];
    else
        tmp[k++] = T[j++];
   while (i<=m) tmp[k++] = T[i++];
   while (j<=b) tmp[k++] = T[j++];
   for (j=0; j<k; j++)
    T[a+j] = tmp[j];

}  // fin tri_fus
```

Si nous voulons construire une solution non récursive, en remontant dans les appels récursifs, on arrive à l'algorithme suivant :
Au départ, on considère n sous listes formées d'un seul élément chacune, donc triées.
Ensuite on fusionne deux à deux des sous-listes de taille 1, cela donne des sous-listes triées de taille 2.
Puis on fusionne deux à deux des sous-listes de taille 2, cela donne des sous-listes triées de taille 4.
On répète le même procédé de fusion jusqu'à l'obtention d'une seule liste triée.

II.A.2.2. Multiplication de grands entiers

Soient X et Y deux grands nombres ayant n chiffres. (n suffisamment grand).
Pour calculer le produit de X par Y on peut programmer l'algorithme que l'on a appris en primaire et qui consiste à multiplier chaque chiffre de Y par tous les chiffres de X en sommant les résultats intermédiaires :
Posons $x_n\ x_{n-1}\ \dots\ x_3\ x_2\ x_1$ les chiffres, du poids fort au poids faible, de X
et de même $y_n\ y_{n-1}\ \dots\ y_3\ y_2\ y_1$ les chiffres représentant Y
La méthode habituelle demande n^2 multiplications entre chiffres et n additions de résultats intermédiaires avec des décalages vers la droite à chaque itération :

			x_n	x_{n-1}	...	x_3	x_2	x_1
×			y_n	y_{n-1}	...	y_3	y_2	y_1
			y_1*x_n	y_1*x_{n-1}		y_1*x_3	y_1*x_2	y_1*x_1
+		y_2*x_n	y_2*x_{n-1}	y_2*x_3		y_2*x_2	y_2*x_1	0
+	y_3*x_n	y_3*x_{n-1}	y_3*x_3	y_3*x_2		y_3*x_1	0	0
+	....		....		....	0	0	0
=	...	....		....		....		....

Les additions et décalages demandent un nombre d'itérations proportionnel à n, donc cette méthode (habituelle) a une complexité de $O(n^2)$.

Une solution " Diviser Pour Résoudre" à ce problème consiste à scinder X et Y en deux entiers de n/2 chiffres chacun :
Soient A la première moitié de X (les n/2 chiffres de poids fort) et B la seconde moitié

de X (chiffres de poids faible).
Soient C la première moitié de Y (les n/2 chiffres de poids fort) et D la seconde moitié de Y (chiffres de poids faible).

Donc on peut écrire : $X = A\ 10^{n/2} + B$ et $Y = C\ 10^{n/2} + D$
Ainsi le produit X Y est égale à : $(A\ 10^{n/2} + B) * (C\ 10^{n/2} + D)$
$= AC\ 10^{n} + (AD + BC)10^{n/2} + BD$

C'est une solution récursive, où la multiplication de deux grands nombres (XY) de taille n est exprimée en fonction d'autres multiplications de grands nombres (AC, AD, BC et BD) de tailles plus réduites (n/2).

Pour réaliser cette solution, on aura alors besoin de faire :
4 multiplications de n/2 chiffres (AC, AD, BC, BD), par des appels récursifs.
3 additions en O(n).
2 décalages (multiplication par 10^n et $10^{n/2}$) en O(n) aussi.

L'équation de récurrence d'un tel algorithme récursif serait donc :

$T(1) = a$	si n=1	(cas particulier)
$T(n) = 4T(n/2) + b\ n$	sinon	(cas général)

On peut trouver la solution par substitution ou bien en utilisant la technique des équations différentielles en posant par exemple $n = 2^k$, ce qui donne l'équation de récurrence : $t_k - 4t_{k-1} = b\ 2^k$ dont la solution est connue $O(4^k)$ et donc $O(n^2)$.
Cette solution (DPR) n'est pas plus efficace que la méthode habituelle (même complexité).

En diminuant le nombre de multiplications (donc le nombre d'appels récursifs), on peut diminuer la complexité de l'algorithme DPR :
Remarquons que $X\ Y = AC10^{n} + [\ (A-B)(D-C) + AC + BD]10^{n/2} + BD$
On aura alors à faire :
3 multiplications (AC, BD, (A-B)(D-C)) par des appels récursifs,
4 additions en O(n)
2 soustractions en O(n) : A-B et D-C
2 décalages en O(n)

L'équation de récurrence devient alors :
$T(n) = 3T(n/2) + c\ n$
dont la Solution est en $O(n^{1.59})$, avec $1.59 \approx \log_2 3$, ce qui est meilleur que $O(n^2)$

Cette dernière méthode est asymptotiquement plus efficace (pour les grands nombres) mais beaucoup moins limpide d'un point de vue pédagogique.

Donnons l'algorithme, sous forme de fonction récursive, correspondant à cette dernière

méthode :

```
Fonction Mult(X, Y, n)
Si n = 1    Retourner X*Y
Sinon
      A ← n/2 premiers chiffres de X
      B ← n/2 derniers chiffres de X
      C ← n/2 premiers chiffres de Y
      D ← n/2 derniers chiffres de Y
      S1 ← Moins(A,B);          // calcule A-B
      S2 ← Moins(D,C)           // calcule D-C
      m1 ← Mult(A,C, n/2)       // calcule AC
      m2 ← Mult(S1,S2, n/2)     // calcule (A-B)(D-C)
      m3 ← Mult(B,D, n/2)       // calcule BD
      S3 ← Plus( m1,m2 )        // calcule [AC] + [ (A-B)(D-C) ]
      S4 ← Plus( S3,m3 )        // calcule [ AC+(A-B)(D-C) ] + [BD]
      P1 ← Décalage(m1, n)      // calcule  AC 10^n
      P2 ← Décalage( S4 )       // calcule [AC+(A-B)(D-C)+BD] 10^(n/2)
      S5 ← Plus( P1,P2 )        // calcule  [AC 10^n] + [ (AC+(A-B)(D-C)+BD) 10^(n/2)]
      S6 ← plus( S5,m3 )        // calcule [AC 10^n+(AC+(A-B)(D-C)+BD) 10^(n/2)] + [BD]
      Retourner S6
Fsi
```

II.A.2.3. Organisation d'un tournoi circulaire :

Organisation d'un tournoi comprenant $n=2^k$ joueurs (ou équipe). Chaque joueur doit affronter chaque autre joueur une seule fois à raison d'un match par jour. Donc la durée du tournoi est de (n-1) jours.

Pour élaborer une solution pour n joueurs, on essaie d'élaborer une solution pour n/2 joueurs. De la même façon, pour élaborer une solution pour n/2 joueurs, on essaie d'élaborer une solution pour n/4. Et ainsi de suite, jusqu'à atteindre 2 joueurs où la solution est triviale.

n = 2

	1
1!	2
2!	1

Pour **n = 4 = 2^2** , on peut construire le tableau

```
      1    2    3
1!    2  ! 3    4
2!    1  ! 4    3
      ----------------
3!    4  ! 1    2
4!    3  ! 2    1
```

Et pour **n = 8 = 2^3** on obtient le tableau

```
      1    2    3    4    5    6    7

1!    2    3    4  ! 5    6    7    8
2!    1    4    3  ! 6    7    8    5
3!    4    1    2  ! 7    8    5    6
4!    3    2    1  ! 8    5    6    7
      ---------------------------------------------
5!    6    7    8  ! 1    4    3    2
6!    5    8    7  ! 2    1    4    3
7!    8    5    6  ! 3    2    1    4
8!    7    6    5  ! 4    3    2    1
```

Etc.

Technique :

Pour k = 1 (ou n=2), la construction de la table est triviale.

A l'étape k (k>1), on construit la table comme suit :
- Partie supérieure gauche : copie de la table de l'étape k-1.
- Partie inférieure gauche : Ajouter 2^{k-1} à chaque élément de la partie supérieure gauche.
- Partie supérieure droite : Au jour 2^{k-1} mettre les indices 2^{k-1}+1, 2^{k-1}+2, ...2^{k-1}+k. Avec rotations circulaires vers le haut pour les autres jours.
- Partie inférieure droite : Au jour 2^{k-1} mettre les indices 1, 2, ...k. Avec rotations circulaires vers le bas pour les autres jours.

II.A.2.4. Tour de Hanoi

Pour certains problèmes, il est beaucoup plus facile de rechercher une solution récursive plutôt qu'une itérative. Le problème des tours de Hanoi constitue un très bon exemple comme nous allons le montrer ci-après.

Enoncé du principe général.

On dispose de 3 tours (ou piquets) A, B et C. Initialement, n disques de diamètres différents sont placés sur A. Un disque plus grand ne doit jamais se trouver sur un disque plus petit. Le problème est de transférer les n disques de A vers C, en utilisant B comme piquet intermédiaire en respectant les deux règles suivantes :

- à un moment donné, seul le disque au sommet d'un piquet peut être transféré vers un autre.
- un disque plus grand ne doit pas se trouver sur un disque plus petit.

Il est très difficile de trouver une solution itérative (essayer...). Par contre, il existe une solution récursive de type DPR, simple et élégante.

Supposons que nous avons une solution pour n-1 disques. Nous pourrons alors définir une solution pour n disques au moyen de la solution de n-1 disques. C'est une décomposition du problème du déplacement de n disques en 2 sous-problèmes de taille n-1 chacun.

Pour n=1 on a un cas trivial : transférer l'unique disque de A vers C.

Sinon, pour transférer n disques (n>1) de A vers C, en utilisant B comme auxiliaire, on procède comme suit:
1- Transférer les n-1 (premiers) disques du sommet A vers B en utilisant C comme intermédiaire.
2- Transférer le disque restant de A vers C.
3- Transférer les n-1 disques de B vers C avec A comme auxiliaire.

Nous remarquons que c'est simple car nous venons de développer une solution pour le cas trivial (n=1) et une solution pour le cas général (n>1) par décomposition en deux sous-problèmes plus simples (le transfert de n-1 disques).

L'algorithme est alors le suivant:

```
Procedure TourdeHanoi( n : entier; de, vers, aux : caractères);
Si n = 1   {cas trivial }
      Ecrire ("Transfert de l'unique disque de", de, "vers", vers)
Sinon
      // Transférer les n-1 disques  au sommet de A vers B avec C comme auxiliaire
      TourdeHanoi(n-1, de, aux, vers)

      Ecrire("Transférer le disque de", de, "vers", vers)

      // Transférer les n-1 disques  de B vers C, utilisant A comme auxiliaire
      TourdeHanoi(n-1, aux, vers,     de)
Fsi
```

- Complexité de TourdeHanoi

Si T(n) est le temps d'exécution d'un appel pour transférer n disques, alors on peut écrire:
T(n) = a si n=1
T(n) = 2 T(n-1) + b sinon
C'est une équation non homogène $t_n - 2t_{n-1} = b$ d'équation caractéristique (x-2)(x-1)=0
Donc la solution générale est $T(n) = C1\ 2^n + C2$ donc en $O(2^n)$

Remarque :

Si nous voulons construire un algorithme non récursif, en remontant dans les appels, on aboutit à l'algorithme suivant :

Imaginons que les piquets sont disposés en triangle. A tous les coups de rang impair, on déplace le plus petit disque sur le piquet suivant dans le sens des aiguilles d'une montre.
A tous les coups de rang pair, on effectue le seul déplacement possible permis n'impliquant pas le plus petit disque.

II.A.3 Recommandations générales sur « Diviser pour Résoudre »

1. Equilibrer les sous-problèmes

La division d'un problème en sous problèmes de taille sensiblement égales contribue de manière cruciale à l'efficacité de l'algorithme.

Considérer le tri par insertion.
On suppose T[1..K] trié et on insère T[K+1] par décalage

```
Tri(n) :
Si n = 1 : Retourner T
Sinon
      Retourner  Insère (Tri (n-1), T[n])
Fsi
```

Insère (T,x) est une procédure qui insère l'élément x dans un tableau ordonné T (en O(n), car n itérations suffisent pour réaliser les décalages et l'insertion du nouvel élément).

Le problème est donc divisé en deux sous problèmes de tailles différentes, l'un de taille 1 et l'autre de taille n-1.

Posons T(n) le temps d'exécution d'un appel à la procédure récursive pour trier un tableau de n éléments.

L'équation de récurrence donnant le temps d'exécution en fonction de n est :
T(n) = a si n=1
= T(n-1) + bn sinon

C'est une forme non homogène ($t_n - t_{n-1} = bn$) d'équation caractéristique : $(x-1)(x-1)^2=0$
La solution générale est donc $T(n) = C1 + C2\ n + C3\ n^2$

Ce qui conduit à $O(n^2)$ (plus complexe que O(n*log n))

Le tri par fusion subdivise le problème en deux sous problème de taille n/2. Ce qui donne une complexité égale à O(n*log(n)).

Voici une implémentation en langage C, du tri par insertion en version récursive :

```
/* Tri par insertion (version récursive)          * /
void tri_ins(int n)  // le tableau T est une variable globale
{
   int i, stop, v;
   // Cas particulier...
   if (n == 1) return;

   // Cas général...
   tri_ins(n-1);

   // Décalages et insertion du dernier élément du tableau
   i = n-2;
   stop = 0;
   v = T[n-1];
   while (i >= 0 && !stop) {
    if (v >= T[i]) {
         T[i+1] = v;
         stop = 1;
    }
    else {
         T[i+1] = T[i];
         i--;
    }
   }
   if (!stop) T[0] = v;
}  // fin tri_ins
```

2. Le nombre de sous problèmes doit être indépendant de n

La décomposition doit générer un nombre de sous problèmes constant et indépendants de n. Sinon la solution ne sera pas efficace.

Exemple : Calcul du déterminant d'une matrice par une décomposition récursive.

$$\begin{aligned} Det(M) &= M[1,1] && \text{Si } n=1 \text{ (taille de la matrice)} \\ &= \sum_{i=1..n} (-1)^{i+1} * M[1,i] * Det(sousMat(M,1,j)) && \text{Si } n > 1 \end{aligned}$$

avec 'sousMat(M,i,j)' désignant la sous-matrice de M, sans la ligne i et sans la colonne j

Pour une matrice de taille n x n, cette décomposition génère n sous-problèmes (sous-matrices de taille n-1 x n-1).

Chaque appel de la fonction récursive avec un paramètre n, génère donc n appels récursifs ainsi que le calcul d'une somme de n éléments. L'équation de récurrence (dans le cas général) donnant le temps d'exécution d'un appel est donc :
$T(n) = n\,T(n-1) + bn$
Le cas particulier étant par exemple $T(0) = a$

On peut résoudre cette équation par substitution :

$$\begin{aligned} T(n) &= n\,T(n-1) + bn = n\,[(n-1)\,T(n-2) + b(n-1)] + bn \\ &= n(n-1)\,T(n-2) + b(n-1)n + bn = n\,(n-1)\,[(n-2)T(n-3)+b(n-2)] + b(n-1)n + bn \\ &= n(n-1)(n-2)\,T(n-3) + bn(n-1)(n-2) + bn(n-1) + bn \\ &\dots \\ &= n!\,T(0) + b\,n!\,[1/1! + 1/2! + 1/3! + \dots 1/(n-1)!\,] \\ &= a\,n! + b\,n!\,[e-1] \rightarrow O(n!) \end{aligned}$$

Alors qu'il existe d'autres algorithmes (Gauss-Jordan) pour le calcul du déterminant d'une matrice carré de taille n en $O(n^3)$. Donc la décomposition DPR appliquée à ce problème donne un très mauvais résultat.

II.B) Approche ascendante : Programmation Dynamique

II.B.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.

L'idée de base dans la programmation dynamique, est d'éviter de calculer deux fois la même chose, en utilisant généralement, une table de résultats déjà calculés, remplie au fur et à mesure que l'on résout les sous problèmes.

C'est donc une méthode ascendante : On commence 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 celui de la recherche du plus court chemin entre deux sommets d'un graphe.

II.B.2. Applications

II.B.2.1. Calcul de C_n^p

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

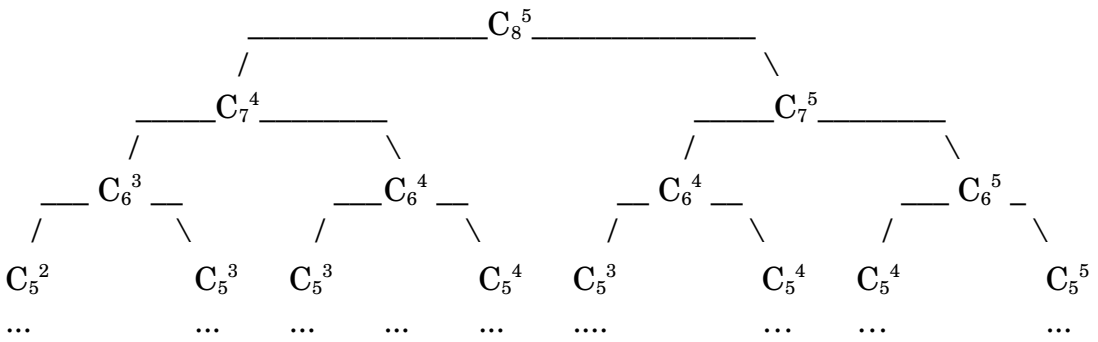
$C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$
$C_0^0 = C_n^0 = C_n^n = 1$

a) L'algorithme récursif (c'est l'approche descendante : Diviser Pour Résoudre) est

comme suit :

```
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 $\Omega(C_n^p)$.

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

Le calcul se fait ligne par ligne et de gauche à droite, c'est à dire que les sous exemplaires les plus simples (pour n et p petits) sont calculés en premiers et les résultats, que l'on sauvegarde dans la table, servent à calculer les exemplaires plus grands.

On peut par exemple remplir la table ligne par ligne et de gauche à droite. Le calcul des éléments d'une ligne i utilise les éléments déjà calculés de la ligne i-1.

```
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
        Fpour
Fpour
Retourner Mat[n,p]
```

Cet algorithme est en O(np).

II.B.2.2. La suite de Fibonacci

La définition de cette suite est donnée par la décomposition :

$f_0 = 0$
$f_1 = 1$
$f_n = f_{n-1} + f_{n-2}$ pour $n > 1$

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

```
Fib_Rec(n:Entier) : Entier;
Si n<=1 :
        Retourner n
Sinon
        Retourner 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;          // Fib(0)
i ← 1;          // Fib(1)
Pour k=1, n :
        j ← i+j;
        i ← j-i
Fpour
Retourner j
```

Cet algorithme est en O(n).

II.B.2.3. Série mondiale

Soient A et B deux équipes qui disputent une série de matchs. L'équipe gagnante est celle qui remporte n victoires en premier. Donc, au maximum 2n-1 matchs car il n'y a pas de match nul.

La probabilité que A remporte une victoire, lors d'un match, est : q1
La probabilité que B remporte une victoire est : q2 = 1-q1

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

On a ainsi une décomposition récursive (DPR) :

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 (le cas général)

La probabilité que A remporte la série à une étape donnée où il lui reste encore i victoires à remporter et il reste à B j victoires (P(i,j)) dépend de l'issue du prochain match à jouer :

- Soit il sera remporté par A auquel cas, il lui restera i-1 victoires supplémentaires à remporter (q1*P(i-1, j)),
- Soit il sera remporté par B et dans ce cas il restera toujours pour A, i victoires supplémentaires à remporter alors qu'il ne restera à B que j-1 victoires supplémentaires (q2*P(i, j-1))

Algorithme récursif :

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

Pour simplifier le calcul de la complexité de cette fonction, ramenons-nous à une formulation utilisant une seule variable :
T(k) = temps d'exécution de P(i, j) avec k = i+j

L'équation de récurrence est $T(k) = 2T(k-1) + b$, dont la solution est $T(k) = C_1 2^k + C_2$

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

Elle est réalisée en utilisant une table remplie diagonale inverse par diagonale inverse. La table ci-dessous montre un exemple de remplissage avec q1= q2 = 0,5:

	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] ← q1*P[k-1, S-k] + q2*P[k, S-k-1]
        FP
FP
```

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

II.B.2.4. Les plus courts chemins

Il s'agit de trouver les plus courts chemins entre chaque paire de sommets d'un graphe pondéré (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 de chemin entre A et C et entre C et B doivent forcément être optimales aussi.

Dans un graphe formé par n sommets, l'algorithme consiste à calculer les plus courts chemins entre chaque paire de sommets en utilisant comme sommets intermédiaires, dans l'ordre et de façon successive 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).

II.B.2.5. 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 nécessaires pour multiplier une série de n matrices : $M_1\ M_2\ M_3\ \ldots\ M_n$. Appelons 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 : $M_i\ M_{i+1}\ M_{i+2}\ ...\ M_j$ 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 sont données par D[i-1] (nombre de lignes) et D[i] (nombre 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 nécessaires 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 optimale (soit m(1,i) opérations 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 optimale (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 le découpage optimal doit se faire au niveau de la $k^{ième}$ matrice, on pourra alors écrire : m(1,n) = m(1,k) + m(k+1,n) + D[0]*D[k]*D[n]
Donc pour trouver la solution optimale, on doit faire varier k de 1 à n-1 pour choisir la plus petite valeur de m(1,n) :

$$m(1,n) = \min_{1 \le k \le n-1} \{m(1,k) + m(k+1,n) + D[0]*D[k]*D[n]\}$$

Dans le cas général, on calcule m(i,j) comme suit :

$$m(i,j) = \min_{i \le k \le j-1} \begin{cases} m(i,i)+m(i+1,j)+D[i-1]*D[i]*D[j] & \text{pour: } M_i(M_{i+1}\ M_{i+2}...M_j) \\ m(i,i+1)+m(i+2,j)+D[i-1]*D[i+1]*D[j] & \text{pour: } (M_i\ M_{i+1})(M_{i+2}...M_j) \\ \quad \dots & \dots \\ m(i,k)+m(k+1,j)+D[i-1]*D[k]*D[j] & \text{pour: } (M_i...M_k)(M_{k+1}...M_j) \\ \quad \dots & \dots \\ m(i,j-1)+m(j,j)+D[i-1]*D[j-1]*D[j] & \text{pour: } (M_i\ M_{i+1}...M_{j-1})M_j \end{cases}$$

avec le cas particulier : m(i,i) = 0

La fonction suivante calcule le terme m(i,j) en utilisant la décomposition ci-dessus. C'est une solution de type Diviser Pour Résoudre, très inefficace à cause du nombre élevé de sous problèmes identiques générés par la décomposition :

```
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) }
        Fpour
        Retourner x
Fsi
```

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] que l'on remplit au fur et à mesure que l'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
        m[i,i] = 0      /* car il n'y a pas de produit de matrices */
Fpour
Pour i:=1,n-1
        m[i,i+1] = D[i-1]*D[i]*D[i+1]      /* Pour le produit : Mi * Mi+1 */
Fpour
Pour s:=2,n-1
        Pour i:=1,n-s
                m(i,i+s) =   min i≤k≤i+s-1 { m(i,k) + m(k+1,i+s) + D[i-1]*D[k]*D[i+s] }
        Fp
Fpour
Retourner m[1,n].
```