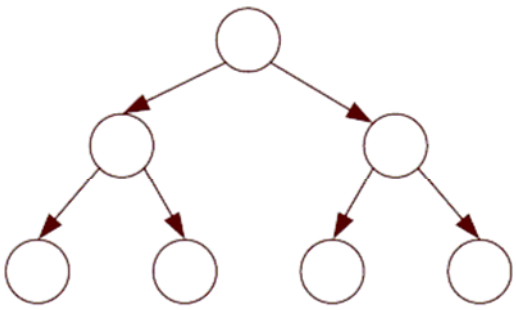
Chapitre 2: La programmation dynamique

1. Introduction

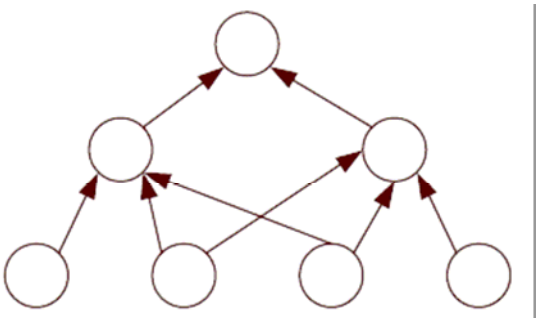
La programmation dynamique est un paradigme de conception qu'il est possible de voir comme une amélioration ou une adaptation de la méthode diviser et régner. Ce concept a été introduit par Bellman, dans les années 50, pour résoudre typiquement des problème d'optimisation.

Pour la petite histoire, Bellman a choisi le terme programmation dynamique dans un souci de communication : son supérieur ne supportait ni le mot « recherche » ni celui de « mathématique ». Alors il lui a semblé que les termes « programmation » et « dynamique » donnaient une apparence qui plairait à son supérieur. En réalité, le terme programmation signifiait à l'époque plus planification et ordonnancement que la programmation au sens qu'on lui donne de nos jours. En un mot, la programmation dynamique est un ensemble de règles que tout un chacun peut suivre pour résoudre un problème donné.

La programmation dynamique est similaire à la méthode diviser et régner en ce sens que, une solution d'un problème dépend des solutions précédentes obtenues des sous-problèmes. La différence significative entre ces deux méthodes est que la programmation dynamique permet aux sous-problemes de se superposer. Autrement dit, un sous-problèmes peut être utilisé dans la solution de deux sous-problèmes différents. Tandis que l'approche diviser et régner crée des sous-problèmes qui sont complètement séparés et peuvent être résolus indépendamment l'un de l'autre. Une illustration de cette différence est montrée par la Figure 5.1. Dans cette figure, le problème à résoudre est à la racine, et les descendants sont les sous-problèmes, plus faciles à résoudre. Les feuilles de ce graphe constituent des sous-problèmes dont la résolution est triviale.
Dans la programmation dynamique, ces feuilles constituent souvent les données de l'algorithme. La différence fondamentale entre ces deux méthodes devient alors claire: les sous-problèmes dans la programmation dynamique peuvent être en interaction, alors dans la méthode diviser et régner, ils ne le sont pas.



diviser et régner



programmation dynamique

Une seconde différence entre ces deux méthodes est, comme illustré par la figure ci-dessus, est que la méthode diviser et régner est récursive, les calculs se font de haut en bas. Tandis que la programmation dynamique est une méthode dont les calculs se font de bas en haut : 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.

Illustration 1: On débute nos exemples par celui du calcul des nombres de Fibonacci. Le problème est de calculer le n premiers nombres de Fibonacci donnés par la formule suivante :

$$F(0) = 1; F(1) = 1;$$
$$F(n) = F(n-1) + F(n-2)$$

L'algorithme implantant cette formule est alors comme suit :

Algorithme 1

```
int  function Fibo(int n){

     if (n <= 1)
        return 1 ;
     else return(Fibo(n-1)+Fibo(n-2))
}
```

Nous avons vu que la complexité de cet algorithme est en exponentielle. La raison de cette inefficacité est due à la multiplicité de calcul d'un même nombre, comme le montre la figure ci-dessous sur n =4.

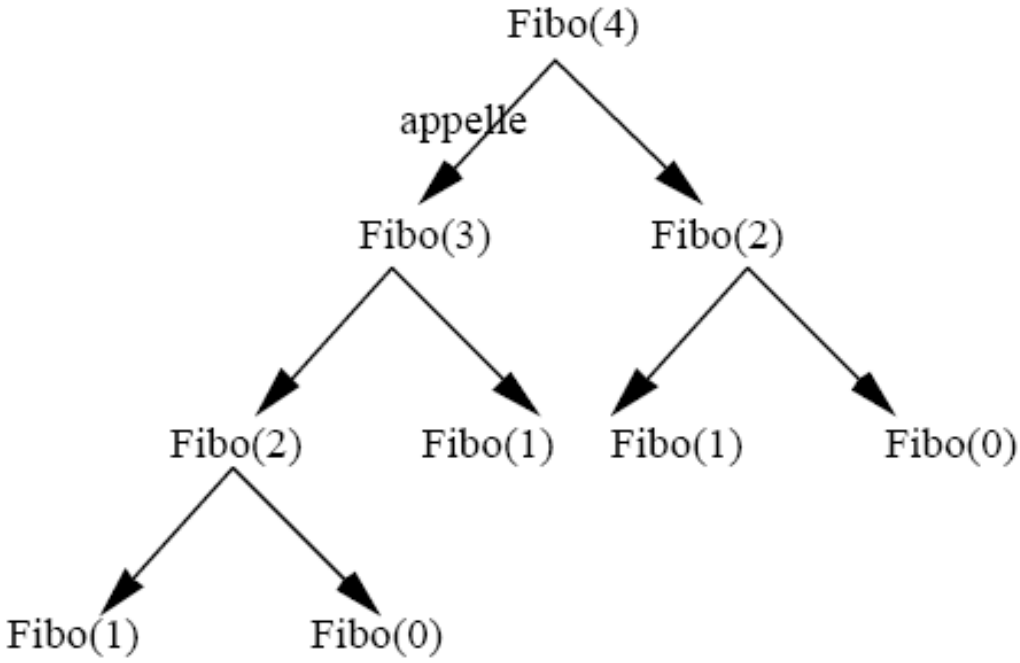


Figure 5.2.

La clef à une solution plus efficace est d'éviter la multiplicité de résolution du même sous-problème. On améliore de loin la complexité temporelle si, une fois calculé, on sauvegarde un résultat, par exemple dans une table. Et au besoin, on le prend de cette table. Cette remarque nous amène à la solution suivante :

Algorithme 2

```
int  function Fib(int n){

if (Fib(n) solution est dans la table)
   return table[n] ;
if (n<= 2)
   return 1;
else {
       sol = Fib(n-1) + Fib(n-2)
       sauvegarder sol dans table comme solution à Fib(n) ;
       return (sol) ;
      }
}
```

Cette approche de résolution est connue sous le nom de fonctions à mémoire, qui est très liée à la programmation dynamique. On y reviendra un peu loin. En suupprimant la récursivité, nous écrivons cet algorithme dans une forme typique de la programmation dynamique.

Algorithme 3

```
int  function Fib(int n){
F[1] =1 ; F[2] =1 ;
For (i=2 ; i<=n ; i++)
   F[i] = F[i-1] + F[i-2];
return (F[i]);
}
```

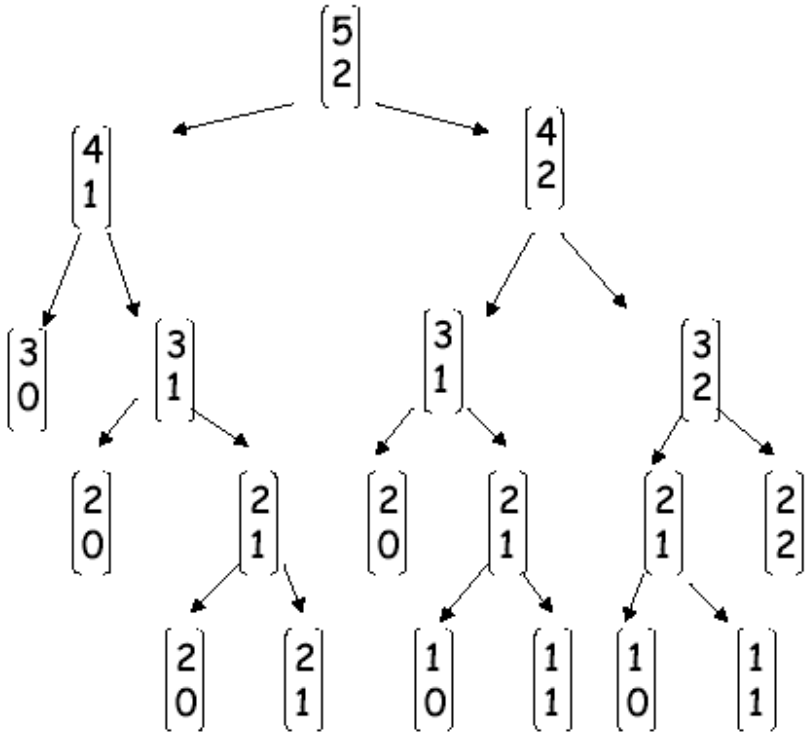
Illustration 2 : Avançons un peu plus dans ce concept en prenant un autre exemple qui est celui du calcul du coefficient binomial.

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & \text{autrement} \end{cases}$$

Si on implante directement cette expression sous cette forme, on obtient la fonction suivante :

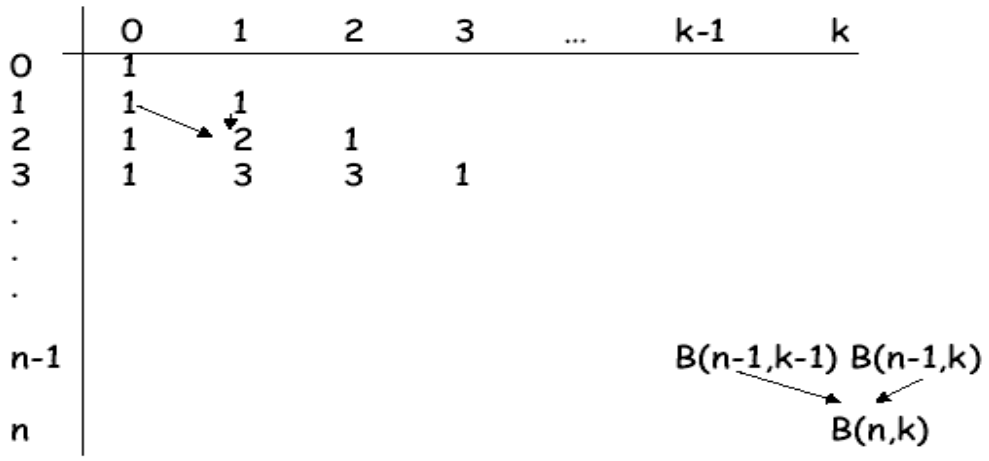
```
int function B(int n,k) {
     if (k = = 0) || (k = = n)
        return 1
     else return(B(n-1,k-1) + B(n-1,k);
} \\ fin de fonction
```

Voyons voir l'exécution de cette fonction sur un exemple de données: n = 5 et k = 2. Remarquez le nombre de fois, par exemple que, le terme $\binom{2}{1}$ est calculé.



Exercice: montrer que la complexité temporelle de cette fonction est en $\theta\binom{n}{k}$.

Pour éviter de calculer plusieurs fois un nombre, l'idée est de créer un tableau où on calcule tous les nombres de petites tailles, ensuite, de tailles de plus en plus grandes avant d'arriver au nombre désiré. Pour ce faire, on procède comme suit :



Pour calculer donc B(n,k), on procède comme suit:

```
for(i=0 ; i<=n ; i++)
    for (j=0; j<=min{i,k}; j++)
        if (i==0) ||(j==i)
            B[i,j] = 1;
        else B[i,j] =  B[i-1,j-1] + B[i-1,j]
return (B[n,k]);
```

On remplit le tableau B ligne par ligne comme suit

Complexité de l'algorithme : cette complexité est réduite au remplissage de la moitié du tableau B. Comme le nombre d'éléments de B est de $k \times n$, la complexité de l'algorithme est par conséquent en $O(kn)$.

Remarque : Il est intéressant de constater que seule la moitié de la matrice B est utilisée pour calculer B[n,k]. Autrement dit, un tableau à une dimension suffirait pour faire ces calculs. On obtient alors l'algorithme suivant :

```
1  unsigned int Binom (unsigned int n, unsigned int m)
2  {
3      Array<unsigned int> b (n + 1);
4      b [0] = 1;
5      for (unsigned int i = 1; i <= n; ++i)
6      {
7          b [i] = 1;
8          for (unsigned int j = i - 1U; j > 0; --j)
9              b [j] += b [j - 1U];
10     }
11     return b [m];
12 }
```

2. Quand et comment utiliser la méthode de la programmation dynamique

La programmation est un outil général de résolution de problèmes. Toutefois, il n'y pas de règle pour affirmer que la programmation dynamique peut ou ne peut être utilisée pour résoudre tel ou tel problème.
Le gros du travail, si on veut utiliser cette méthode, réside tout d'abord dans l'obtention de l'expression récursive de la solution en fonction de celle des sous-problèmes (de taille plus petite). Notons que dans les problèmes d'optimisation, cette manière d'obtenir la solution optimale à partir des solutions optimales des sous problèmes s'appelle **le principe d'optimalité de Bellman**. Il est important de souligner que ce principe, bien que naturel, n'est pas toujours applicable.

Exercice : Trouver un exemple où ce principe n'est pas applicable.

Une fois cette expression obtenue, on analyse ce qui se passe dans une implantation récursive naïve : si on se rend compte que la solution de même problèmes est calculée plusieurs fois, on est alors dans le cadre de la programmation dynamique. Le découpage du problème devrait naturellement conduire à la définition de la table (qui peut être de dimension 1,2,3, …). Remarquez qu'une case de la table correspond à un sous-problème. Par ailleurs, le nombre de sous-problèmes peut être très grand. La complexité obtenue, de l'algorithme de programmation dynamique, n'est pas forcément polynomial.

Si on est dans le cadre de la méthode de la programmation dynamique, les étape suivies peuvent être résumées comme suit :

a. **obtention de l'équation récursive** liant la solution d'un problème à celle de sous-problèmes.
b. **initialisation de la table**: cette étape est donnée par les condition initiale de l'équation obtenue à l'étape 1.
c. **remplissage de la table**: cette étape consiste à résoudre les sous-problèmes de taille de plus en plus grandes, en se servant bien entendu de l'équation obtenue à l'étape 1.

d. lecture de la solution : l'étape 3 ne conduit qu'à la valeur (optimale) du problème de départ. Elle ne donne pas directement la solution conduisant à cette valeur. En générale, pour avoir cette solution, on fait un travail inverse en lisant dans la table en partant de la solution finale et en faisant le chemin inverse des calculs effectués en à l'étape 3.

3. Étude de quelques exemples.

Dans cette section, il sera question de voir en action la méthode de la programmation dynamique sur des problèmes essentiellement d'optimisation.

3.1. Déterminer le plus court chemin dans un graphe – algorithme de Floyd

Soit un graphe $G = (X, V)$ ayant X comme ensemble de sommets et V comme ensemble d'arcs. Le poids de l'arc a est un entier naturel noté $l(a)$. La longueur d'un chemin est égale à la somme des longueurs des arcs qui le composent. Le problème consiste à déterminer pour chaque couple (x_i, x_j) de sommets, le plus court chemin, s'il existe, qui joint x_i à x_j.

Nous commençons par donner un algorithme qui détermine les longueurs des plus courts chemins notées $\delta(x_i, x_j)$. Par convention, on note $\delta(x_i, x_j) = \infty$ s'il n'existe pas de chemin entre x_i et x_j (en fait il suffit dans la suite de remplacer ∞ par un nombre suffisamment grand par exemple la somme des longueurs de tous les arcs du graphe). La construction effective des chemins sera examinée ensuite. On suppose qu'entre deux sommets il y a au plus un arc. En effet, s'il en existe plusieurs, il suffit de ne retenir que le plus court.

Les algorithmes de recherche de chemins les plus courts reposent sur l'observation très simple (mais combien importante) suivante:

Remarque

Si f est un chemin de longueur minimale joignant x à y et qui passe par z , alors il se décompose en deux chemins de longueur minimale l'un qui joint x à z et l'autre qui joint z à y.

Dans la suite, on suppose les sommets numérotés $x_1, x_2, ..., x_n$ et, pour tout $k > 0$, on considère la propriété P_k suivante pour un chemin: $P_k(f)$: Tous les sommets de *f*, autres que son origine et son extrémité, ont un indice strictement inférieur à *k*.

On peut remarquer d'une part qu'un chemin vérifie P_1 si, et seulement si, il se compose d'un unique arc. D'autre part la condition P_{n+1} est satisfaite par tous les chemins du graphe. Notons par $\delta_k(x_i, x_j)$ la longueur du plus court chemin vérifiant la propriété P_k et qui a pour origine x_i et pour extrémité x_j. Cette valeur est ∞ si aucun tel chemin n'existe. Ainsi $\delta_1(x_i, x_j) = \infty$ s'il n'y a pas d'arc entre x_i et x_j, et vaut $l(a)$, si *a* est cet arc. D'autre part $\delta_{n+1} = \delta$.

Le lemme suivant permet de calculer les δ_{k+1} connaissant les $\delta_k(x_i, x_j)$. On en déduira un algorithme itératif.

Lemme Les relations suivantes sont satisfaites par les δ_k :

$$\delta_{k+1}(x_i, x_j) = \min(\delta_k(x_i, x_j), \delta_k(x_i, x_j) + \delta_k(x_k, x_j))$$

Preuve

Soit un chemin de longueur minimale satisfaisant P_{k+1}, ou bien il ne passe pas par x_k et on a la relation suivante qui est vérifiée :

$$\delta_{k+1}(x_i, x_j) = \delta_k(x_i, x_j)$$

ou bien il passe par x_k et, d'après la remarque ci-dessus, il est composé d'un chemin de longueur minimale joignant x_i à x_k et satisfaisant P_k et d'un autre minimal aussi joignant x_k à x_j. Ce chemin est donc de longueur : $\delta_k(x_i, x_j) + \delta_k(x_k, x_j)$.

L'algorithme suivant pour la recherche du plus court chemin met à jour une matrice delta[i,j] qui a été initialisée par les longueurs des arcs et par un entier suffisamment grand s'il n'y a pas d'arc entre x_i et x_j. À chaque itération de la boucle externe, on fait croître l'indice *k* du P_k calculé.

```
for k := 1 to n
   for i := 1 to n
      for j := 1 to n
        delta[i,j] := min(delta[i,j], delta[i,k] + delta[k,j])
```

Sur l'exemple du graphe donné sur la Figure 5.3, on part de la matrice δ_1 donnée par

$$\delta_1 = \begin{pmatrix} 0 & 1 & \infty & 4 & \infty & \infty & \infty \\ \infty & 0 & 3 & 2 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & 0 & 2 & \infty & 6 \\ \infty & 3 & \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 2 & \infty & 0 & 1 \\ 4 & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

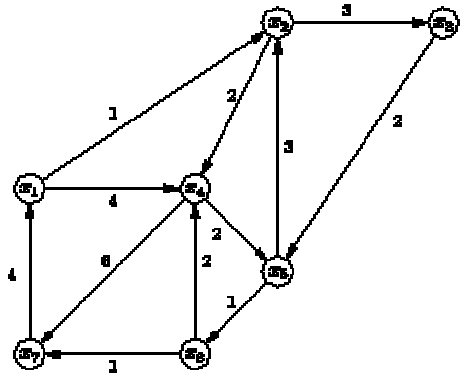


Figure 5.3: Un graphe aux arcs valués

Après le calcul on obtient:

$$\delta = \begin{pmatrix} 0 & 1 & 4 & 3 & 5 & 6 & 7 \\ 10 & 0 & 3 & 2 & 4 & 5 & 6 \\ 8 & 5 & 0 & 5 & 2 & 3 & 4 \\ 8 & 5 & 8 & 0 & 2 & 3 & 4 \\ 6 & 3 & 6 & 3 & 0 & 1 & 2 \\ 5 & 6 & 9 & 2 & 4 & 0 & 1 \\ 4 & 5 & 8 & 7 & 9 & 10 & 0 \end{pmatrix}$$

Pour le calcul effectif des chemins les plus courts, on utilise une matrice qui contient suiv[i,j], le sommet qui suit *i* dans le chemin le plus court qui va de *i* à *j*. Les valeurs suiv[i,j] sont initialisées à *j* s'il existe un arc de *i* vers *j* et à -1 sinon ; suiv[i,i] est lui initialisé à *i*. Le calcul précédent qui a donné δ peut s'accompagner de celui de $Suiv$ en procédant comme suit:

```
for k := 1 to n
   for i := 1 to n
      for j := 1 to n
       if delta[i, j] > (delta[i, k] + delta[k, j]) then
          begin
          delta[i, j] :=  delta[i, k] + delta[k, j];
          suivant[i, j] := suivant[i, k];
          end;
```

Une fois le calcul des deux matrices effectué, on peut retrouver le plus court chemin qui joint *i* à *j* , à l'aide la procédure ci-dessous:

```
procedure PlusCourtChemin(i, j: integer);
   var k: integer;
   begin
   k := i;
   while k <> j do
      begin
      write (k, ' ');
      k := suivant[k, j];
      end;
   writeln(j);
   end;
```

Sur l'exemple précédent, on obtient la matrice suivante:

$$Suiv = \begin{pmatrix} 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 4 & 2 & 3 & 4 & 4 & 4 & 4 \\ 5 & 5 & 3 & 5 & 5 & 5 & 5 \\ 5 & 5 & 5 & 4 & 5 & 5 & 5 \\ 6 & 2 & 2 & 6 & 5 & 6 & 6 \\ 7 & 7 & 7 & 4 & 4 & 6 & 7 \\ 1 & 1 & 1 & 1 & 1 & 1 & 7 \end{pmatrix}$$

3.2: Multiplication chaînée de matrices

Soit n matrices $M_1, M_2, ..., M_n$; chaque matrice M_i possède d_{i-1} lignes et d_i colonnes. Le problème est d'effectuer $M_1 \times M_2 \times ... \times M_n$ en un minimum d'opérations de multiplications.

Rappels

a. Pour faire le produit $M_1 \times M_2$, il est nécessaire que le nombre de colonnes de M_1 soit égal au nombre de ligne de M_2

b. Le nombre de multiplications engendrées par $M_i \times M_j$, de dimension respectivement de $(d_{i-1} \times d_i)$ et $(d_i \times d_j)$ est égal à $d_{i-1} \times d_i \times d_j$.

Parce que la multiplication est une opération associative **(** $M_1(M_2 \times M_3) = (M_1 \times M_2)M_3$, il existe une multitude de manières d'effectuer le produit entre les matrices. Par exemple, soit les trois matrices suivantes :

$$A(13 \times 5) \ ; \ B(5 \times 89) \ ; \ C(89 \times 3); \ D(3 \times 34)$$

Effectuons les calculs des combinaisons de produit suivants :

$M = ((A \times B)C)D$

AB : $13 \times 5 \times 89 = 5785$ multiplications
$(AB)C$: $13 \times 89 \times 3 = 3471$ multiplications
$((AB)C)D$: $13 \times 3 \times 34 = 1326$ multiplications
---
10 582 multiplications

$((AB)C)D$: 10582 multiplications
$(AB)(CD)$: 52201 multiplications
$(A(BC))D$: 2856 multiplications
$A((BC)D)$: 4 055 multiplications
$A(B(CD))$: 26 418 multiplications

Idée 1 : **la méthode brute:** On insère les parenthèses de toutes les manières possibles et ensuite, pour chacune d'elle, on compte le nombre de multiplications engendrées. Procédons comme pour diviser et régner en subdivisant le problème en deux sous-problèmes comme suit :

$$M = (M_1 \times M_2 \times ... \times M_i)(M_{i+1} \times M_{i+2} \times ... \times M_n) \qquad (1)$$

Si $t(n)$ est le coût pour effectuer le produit ci-dessus, alors $t(i)$ va représenter le coût du produit $(M_1 \times M_2 \times ... \times M_i)$ et $t(n-i)$ celui de $(M_{i+1} \times M_{i+2} \times ... \times M_n)$. Alors, pour un i donné, le coût du produit est clairement $t(n-i)\ t(i)$. Comme l'indice *i* de séparation peut être à n'importe laquelle des positions 1,2,…,n-1, alors la relation entre $t(n)$, $t(i)$ et $t(n-i)$ est comme suit :

$$t(n) = \sum_{i=1}^{n-1} t(i)t(n-i)$$

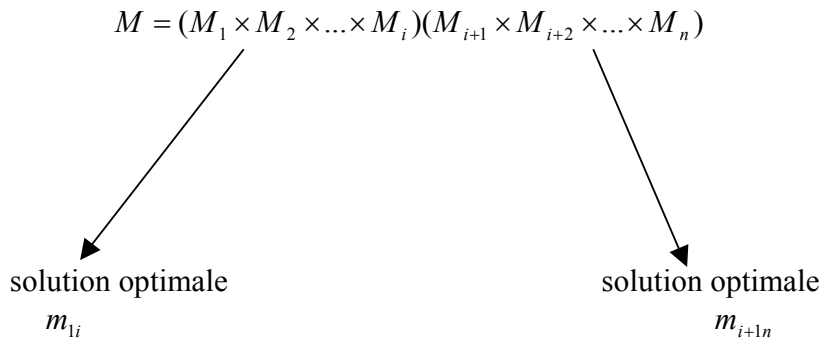
$t(1) = 1$ (pourquoi donc ?)

Exercice : résoudre l'équation récurrente ci-dessus.

Idée 2: la programmation dynamique

1. **caractériser la structure d'une solution optimale**

si $m_{ij}; 1 \le i \le j \le n$: solution optimale pour le faire le produit $(M_{i+1} \times M_{i+2} \times ... \times M_j)$, alors la solution du problème est le calcul de m_{1n}



2. définir récursivement la valeur d'une solution optimale en fonction des solution optimales des sous-problèmes.

Les sous-problèmes : Déterminer le coût minimum d'un parenthétisage de

$$(M_i \times M_{i+2} \times ... \times M_j)$$

i = j : aucune multiplication : $m_{ij} = 0$

i < j : m_{ij} = coût minimum pour calculer le produit $(M_i \times M_{i+2} \times ... \times M_k)$

+

coût minimum pour calculer le produit $(M_{k+1} \times M_{k+2} \times ... \times M_j)$

+

coût pour multiplier les deux matrices $d_{i-1} \times d_k \times d_j$

Comme, on ne sait pas à priori où placer l'indice *k*, on doit essayer toutes les positions et prendre celle qui engendre le moins de multiplications. Cela donne donc :

$$m_{ij} = \begin{cases} 0; i = j \\ \min_{i \le k < j} \{m_{ik} + m_{k+1j} + d_{i-1} d_k d_j; i < j \end{cases}$$

Exercice : implanter l'équation récursive ci-dessus, et déterminer sa complexité temporelle et spatiale.

3. Au lieu de calculer la récurrence directement (autrement dit, d'une manière descendante), on utilise l'approche ascendante de la programmation dynamique comme suit :

Construction de la table diagonale par diagonale. La diagonale s contient les éléments m_{ij} tel que $j - i = s$; pour s= 0,1,2,3,…,n-1, c'est-à-dire pour les différences d'indice de plus en plus grandes.

$$s = 0; \qquad m_{ii} = 0; \qquad i = 1,2,...,n;$$

$$s = 1; \qquad m_{ii+1} = d_{i-1}d_i d_{i+1}; \qquad i = 1,2,...,n-1;$$

$$1 < s < n; \qquad m_{ii+s} = \min_{i \le k < i+s} \{m_{ik} + m_{k+1i+s} + d_{i-1}d_k d_{i+s}\} \qquad i = 1,2,...,n-s;$$

L'algorithme sera alors comme suit :

1. for (i=1; i<=n; i++)
   m[i,i]=0 ;
2. for (i =1; i<=n-1; i++)
   m[i,i+1] = d[i-1]*d[i]*d[i+1];
3. for (s =2; i <= n-1; s++)
   for(i=1; i<= n-s; i++)
   m[i,i+s] = $\min_{i \le k < i+s}$ { m[i,k]+m[k+1,i+s]+d[i-1]*d[k]*d[i+s]};

Complexité : Clairement, elle est en $O(n^3)$: les deux boucles for imbriquées sont exécutées en $O(n^2)$ multipliées par le calcul du minimum qui s'exécute en $O(n)$.

Illustration Reprenons l'exemple précédent à savoir :

$$A(13 \times 5) \ ; \ B(5 \times 89) \ ; \ C(89 \times 3); \ D(3 \times 34)$$

s = 1 : $m_{12} = 5785; \ m_{23} = 1335; \ m_{34} = 9087;$

s = 2 : $m_{13} = \min\{m_{11} + m_{23} + 13 \times 5 \times 3; m_{12} + m_{34} + 13 \times 89 \times 3\} = 1530$

$m_{24} = \min\{m_{22} + m_{34} + 5 \times 89 \times 34; m_{23} + m_{44} + 5 \times 3 \times 34\} = 1845$

s = 3 ; $m_{14} = \min\{m_{11} + m_{24} + 13 \times 5 \times 34; m_{12} + m_{34} + 13 \times 89 \times 34; m_{13} + m_{44} + 13 \times 3 \times 34\} = 2856$.

Pour déterminer l'ordre dans lequel les matrices sont multipliées, on procède comme suit en lisant les valeurs de *k* de la fin vers le début. Ainsi,

m_{14} est donné par k =3 ;
m_{13} est donné par k =2 ;

Ce qui signifie que le produit doit être fait comme suit : $((M_1 \times M_2) \times M_3)M_4$

3.3. Problème du sac à dos en nombres entiers

Nous avons introduit au chapitre précédent le problème du sac à dos. Rappelons la définition de ce problème.

Soit un ensemble de n objets $N = \{1,2,...,n\}$, et un sac à dos pouvant contenir un poids maximal de W. Chaque objet a un poids w_i et un gain v_i. Le problème consiste à choisir un ensemble d'objets parmi les *n* objets, au plus un de chaque, de telle manière que le gain total soit maximisé, sans dépasser la capacité W du sac. Dans cette version que nous présentons dans ce chapitre, un objet est soit choisi soit ignoré. Autrement dit, les objets sont indivisibles, et de ce fait nous ne pouvons pendre une portion d'un objet dans le sac.

En termes mathématiques, nous avons ce qui suit :

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\sum w_i x_i \leq W$$

$$x_i = 0,1.$$

L'intégralité des objets rend ce problème difficile à résoudre. La stratégie vorace proposée dans le chapitre précédent ne fonctionne plus dans ce cas.

Exercice : Donner un exemple où les différentes stratégies voraces auxquelles vous pourriez penser ne génèrent pas la solution optimale.

Exercice : un simple algorithme pour ce problème est de générer toutes les solutions possibles (un objet est soit choisi soit ignoré). Concevoir l'algorithme basé sur cette stratégie. Montrer que sa complexité est en $O(2^n)$

Propriété récursive du problème :

La programmation dynamique peut être développée en divisant le problème en deux sous-problème comme suit :

$P_{i,j}$ désigne le gain maximum généré par le choix des i premiers objets dont la somme des poids ne dépasse pas *W,* alors résoudre le problème revient à trouver la valeur de $P_{n,W}$.

En calculant $P_{i,j}$ la séquences d'objets peut être divisée en deux : les $(i-1)$ premiers objets et l'objet *i.* L'objet *i* est soit choisi soit ignoré dans $P_{i,j}$.

si l'objet *i* est choisi, avant de l'inclure, on doit s'assurer que son poids ne dépasse pas la capacité *j* du sac à dos. Si tel est le cas, alors il contribue à la solution optimale par le gain v_i. Par conséquent, nous avons bien

$$P_{ij} = P_{i-1,j-w_i} + v_i$$

Si l'objet *i* n'est pas choisi dans la solution optimale. Dans ce cas, nous avons la capacité du sac inchangée. Il suffirait donc de trouver la solution optimale parmi les $i-1$ premiers objets, soit $P_{i-1,j}$.

Bien entendu, pour trouver $P_{i,j}$, il suffirait de prendre le maximum entre le cas où l'objet est choisi ou ignoré.

Les cas de base sont : $P_{i,j} = 0$ pour $i = 0$ ou $j = 0$ (pourquoi donc ?)

Cela nous amène aux relations récursives suivantes :

$$P_{ij} = \begin{cases} 0; i = 0 ... ou ... j = 0 \\ p_{i-1,j}; j < w_i; i > 0 \\ \max\{p_{i-1,j}, p_{i-1,j-w_i} + v_i\} \end{cases}$$

L'algorithme de programmation dynamique est alors comme suit :

```
1. For (i=1 ;i<=n ; i++)
     P[i,0] = 0;
2. For (i=1 ;i<=k ; i++)
     P[0,i] = 0;
3. For (i=1 ;i<=n ; i++)
     For (j=1 ;i<=k ; j++){
          P[i,j]=P[i-1,j];
          if (j >= w_i ){
               if (P[i-1,j- w_i ]+ v_i  >P[i-1,j])
               P[i,j] = P[i-1,j- w_i ]+ v_i  ; }
        }
```

Complexité: Il est clair que cette complexité est dominée par les deux boucles « for » imbriquées l'une dans l'autre : une est itérée *k* fois et l'autre est itérée *n* fois. Par conséquent, la complexité de tout l'algorithme est en $O(kn)$.

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 :

	0	1	2	3	4	5	6	7	8	9	10	11
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. 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] = min{P[4,11],P[4,11-w_5]+v_5} = P[4,11]}. Cela signifie que l'objet 5 ne fait partie de la solution optimale.
- Ensuite P[4,11] est donné par P[3, 11-w_4] +v_4 = P[3,5]. Autrement dit, l'objet 4 est choisi.
- Ensuite P[3,5] est donné par P[2,5-w_3 1]+ v_3 = P[2,0]+18. L'objet 3 est choisi.
- En continuant de la sorte, on trouve que P[2,0] = P[1,0]=P[0,0].

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

Exercice : Écrire un algorithme qui détermine la composition de la solution optimale.

3.4 : Problème du voyageur de commerce

Définition : Étant un graphe valué $G = (X, V)$. Le problème du voyageur de commerce consiste, en partant d'un sommet donné, de trouver un cycle de poids minimum passant par tous les sommets une et ne seule fois, et retournant au sommet de départ.

Sans perte de généralité, on identifie les *n* sommets par les entiers $\{1,2,...,n\}$, et on suppose que le cycle commence au sommet 1. La distance entre deux sommets *i* et *j* est notée par d_{ij}.

Pour obtenir l'équation de récurrence résolvant ce problème, procédons comme suit :

Il est clair que tout cycle est constitué d'un arc (1,k) et d'un chemin simple (partant de *k* et passant une et une seule fois par tous les sommets de V-{1,k}.

Soit donc D[i,S] la distance d'un plus court chemin partant de *i*, passant par tous les points de *S,* une et seule fois, et se terminant au sommet 1. La relation suivante n'est donc pas difficile à établir :

$$D[i,S] = \min_{j \in S} \{d_{ij} + D[j, S - \{j\}]\} \quad \text{(a)}$$

La solution optimale est donc donnée par D[1,V-{1}].

$$D[1, V - \{1\}] = \min_{2 \le j \le n} \{d_{ij} + D[j, V - \{1, j\}]\}$$

Il est clair que D[i,∅]= d_{i1}.

Nous avons donc à partitionner l'ensemble V-{1}. Le nombre de sous-ensemble qu'on en générer est donc $2^{n-1} - 1$. Par conséquent, la dimension de la table à construire est $(n-1)$ par $2^{n-1} - 1$.

Par exemple pour n = 4, on aura à construire la table suivante :

	{}	{2}	{3}	{4}	{2,3}	{2,4}	{3,4}
1							
2							
3							

Illustration : Soit donc le graphe suivant :

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

Nous avons donc :

D[2, ∅] = 5 ; D[3, ∅] = 6 ; D[4, ∅] = 8 ;

$D[2,\{3\}] = d_{23} + \text{D}[3, \varnothing] = 15$; $D[2,\{3\}] = 18;$

$D[3,\{2\}] = 18;\ D[3,\{4\}] = 20;$

$D[4,\{2\}] = 13;\ D[4,\{3\}] = 15;$

Ensuite, on calcule $D[i,S]$ avec |S| =2 ; $i \neq 1; i \notin S$.

$$D[2,\{3,4\}] = \min\{d_{23} + D[3,\{4\}]\}, d_{24} + D[4,\{3\}] = 25$$

$$D[3,\{2,4\}] = \min\{d_{32} + D[2,\{4\}]\}, d_{34} + D[4,\{2\}] = 25$$

$$D[4,\{2,3\}] = \min\{d_{42} + D[2,\{3\}]\}, d_{43} + D[3,\{2\}] = 23$$

Finalement, on obtient :

$$D[1,\{2,3,4\}] = \min\{d_{12} + D[2,\{3,4\}], d_{13} + D[3,\{2,4\}], d_{14} + D[4,\{2,3\}]\}$$

$$= \{35,40,43\} = 35.$$

La valeur de la solution optimale est donc 35.

Pour le parcours à faire ayant la valeur optimale de 25, on procède de la manière suivante:

pour chaque $D[i,S]$, on retient l'indice j minimisant le membre droit de l'équation (a), en commençant toujours à partir de la valeur de la solution optimale et en rebroussant chemin jusqu'à trouver la solution complète. Ainsi, dans notre exemple, nous avons

L'indice minimisant $D[1,\{2,3,4\}]$ est 2. Le cycle donc commence par 1 et doit passer par le sommet 2. Le reste du cycle peut être obtenu à partir de $D[2,\{3,4\}]$. L'indice minimisant cette expression est 4. Par conséquent, le cycle devra ensuite passer par le sommet 4. Le reste du cycle peut être obtenu de $D[4,\{3\}]$. L'indice minimisant cette expression ne peut être que 3. Le cycle est donc 1-2-4-3-1.

Complexité : On doit remplir tout le tableau de dimension $(n-1)$ par $2^{n-1}-1$. De Plus, le calcul de chaque D[i,S] nécessite l'examen de |S| autres cases (pour déterminer le minimum). Par conséquent, la complexité de cet algorithme est en $O(n^2 2^n)$.

Exercice : déterminer la complexité spatiale de cet algorithme.

4. Les fonctions à mémoire

Tout comme la méthode diviser et régner calcule des valeurs plusieurs fois, la programmation dynamique peut calculer elle aussi des valeurs inutiles. En effet, on faisant les calculs de bas en haut, on est amener à calculer des valeurs qui peuvent ne pas être utiliser par la suite: on calcule tous les sous-problèmes de petite taille, ensuite de taille un peu plus grande et ainsi de suite, sans se soucier si ces valeurs vont entrer dans le calcul de la valeur optimale du problème de départ. La technique de la fonction mémoire permet de combiner l'élégance de la méthode diviser et

régner (de la récursivité) avec l'efficacité de la programmation dynamique. L'idée donc est d'utiliser un tableau T avec la fonction récursive f exprimant la résolution du problème donné.

```
Fonction f(x1,x2,…,xn)
 {
   si T[x1,x2,…,xn] ≠ la valeur d'initialisation
        alors retourner T[x1,x2,…,xn] ;
   sinon {
            s = f(x1,x2,…,xn) ;
            T[x1,x2,…,xn] = s ;
            retourner s ;
           }
 }
```

De ce fait, on gagne à ne pas calculer inutilement certaines cases.

Remarque : Le premier exemple qu'on a vu est celui du calcul des nombre de Fibonacci.

Prenons comme deuxième exemple celui du voyageur de commerce. Si on veut implanter notre solution d'une manière ascendante, nous rencontrons le problème évoqué au début du ce chapitre : certaines valeurs de D[i,S] sont recalculées plusieurs fois et l'algorithme est très inefficace.

Exercice : Montrer que L,implantation ascendante de la fonction (a) ci-dessus génère une complexité en O(n !).

Pour calculer D[i,S] d'une manière ascendante, nous avons besoin de générer tous les ensembles vides, puis contenant un élément, puis deux, etc. La conception d 'un tel générateur n'est certainement pas difficile mais elle est certainement fastidieuse.

L'utilisation d'une fonction à mémoire consiste à adjoindre à la fonction récursive une table de taille suffisante. Initialement, tous les éléments de cette table ont une valeur spéciale indiquant qu'ils ne sont pas encore définis. Ensuite, chaque fois qu'on appelle la fonction, on regarde dans la table pour voir si cette valeur a été déjà calculée sur les même paramètres. Si c'est le cas, on retourne la valeur stockée dans la table. Autrement, on procède au calcul de la fonction. Avant de retourner la valeur de la fonction, on la stocke dans la table à l'endroit approprié. De cette manière, on n'aura pas à calculer la fonction plus d'une fois. Pour les mêmes valeurs des paramètres. Cela nous donne l'implantation suivante :

On initialise les éléments de tab à –1 (une distance ne peut être négative).

fonction D(int i, set S){

si S = ∅ retourner d_{i1} ;
si tab[i,S] ≥ 0 retourner tab[i,S];
petitshort = un grand nombre;
pour $j \in S$ faire
    distance = $d_{ij} + D(i, S - \{j\})$;
    si distance < petitshort
        petishort = distance ;
    tab[i,S] = petitshort ;
retourner petitshort;

La fonction D allie la clarté de la conception diviser et régner (récursive) avec l'efficacité de la programmation dynamique.

Exercice : Montrer comment calculer le coefficient binomial en utilisant une fonction à mémoire.

Sources

1. G. Brassard, P. Bratley (1996): Fundamentals of algorithmics, Prentice Hall
2. E. Horowitz, S. Sahni, S., Rajasekaran (1997): Computer algorithms, Computer Sciences Press.
3. J.J. Levy: Notes de cours d'algorithmique, École Polytechnique, France.