# PROGRAMMATION DYNAMIQUE

## SOMMAIRE

# INTRODUCTION

L'**optimisation** est une branche des mathématiques cherchant à modéliser, à analyser et à résoudre analytiquement ou numériquement les **problèmes** qui consistent à minimiser ou maximiser une fonction sur un ensemble. Ces problèmes peuvent admettre plusieurs solutions possibles. Chaque solution a une valeur obtenue à partir d'une **fonction de coût** (aussi appelée **fonction objectif**), et on souhaite trouver une solution ayant la valeur optimale (minimale ou maximale). Une telle solution, dite optimale, au problème n'est pas unique, puisqu'il peut y avoir plusieurs solutions qui donnent la valeur optimale.

**L'approche robuste** ou **naïve** pour résoudre un problème d'optimisation consiste à évaluer la **fonction objectif** pour chaque élément de son ensemble de définition, puis choisir la solution dont la valeur est optimale. La principale limite de cette approche est le temps mis pour calculer la valeur de toutes les solutions possibles. Ce temps est généralement **exponentiel** ; ce qui rend l'approche naïve *irréaliste* lorsque le nombre de valeurs possibles pour chaque paramètre, dont dépend une solution, est élevé. Il existe plusieurs méthodes permettant d'aboutir à une solution optimale en un temps raisonnable. La **programmation dynamique** est l'une de ces méthodes. De façon générale, la **programmation dynamique** résout des problèmes d'optimisation en ne calculant que des solutions qui aident à obtenir des valeurs optimales.

Dans ce rapport, **la section I** sera consacrée aux fondamentaux de la programmation dynamique, ensuite **La section II** présentera notre étude de divers exemples de problèmes d'optimisation solutionnés par la programmation dynamique. Pour continuer**, La section III** nous fournira des outils de comparaison de cette méthode face à d'autres méthodes algorithmiques. Enfin dans un souci d'initiation à la recherche, on verra dans **la section IV** différentes applications de la programmation dynamique dans la vie courante.

# I. LA PROGRAMMATION DYNAMIQUE, QU'EST-CE QUE C'EST ?

## 1. DEFINITION ET HISTORIQUE

En informatique, la **programmation dynamique** est une méthode algorithmique pour résoudre des problèmes d'optimisations; À l'époque, le terme « programmation » signifiait planification et ordonnancement. Elle 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.

R. Bellman (1920-1984)

Le terme *programmation dynamique* était utilisé dans les **années 1940** par Richard Bellman pour décrire le processus de résolution de problèmes où on trouve les meilleures décisions les unes après les autres. En 1953, il en donne la définition moderne, où les décisions à prendre sont ordonnés par sous-problèmes et le domaine a alors été reconnu par **IEEE(**Institute of Electrical and Electronics Engineers**)** comme un sujet d'analyse de systèmes et d'ingénierie. La programmation dynamique a connu un grand succès, car de nombreuses fonctions économiques de l'industrie étaient de ce type, comme la conduite et l'optimisation de procédés chimiques, ou la gestion de stocks. La contribution de Bellman est connue sous le nom d'équation de Bellman, qui présente un problème d'optimisation sous forme récursive.

## 2. PRINCIPE DE LA PROGRAMMATION DYNAMIQUE

La programmation dynamique s'appuie sur le principe d'optimalité de Bellman qui stipule ***qu'une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous problèmes***.

Il existe deux méthodes pour calculer effectivement une solution : la méthode ascendante et la méthode descendante.

- Dans la méthode ascendante, on commence par déterminer des solutions pour les sous-problèmes élémentaires; puis, de proche en proche, on détermine les solutions des problèmes en utilisant le principe d'optimalité et on mémorise les résultats dans un tableau.
- Dans la méthode descendante, on écrit un algorithme récursif mais on utilise la mémoïsation (qui est une technique d'optimisation de code dont le but est de diminuer le temps d'exécution d'un programme informatique en mémorisant les valeurs retournées par une fonction) pour ne pas résoudre plusieurs fois le même problème.

La conception d'un algorithme de programmation dynamique est typiquement découpée en quatre étapes :

1. Caractériser la structure d'une solution optimale.
2. Définir (souvent de manière récursive) la valeur d'une solution optimale.
3. Calculer la valeur d'une solution optimale.
4. Construire une solution optimale à partir des informations calculées.

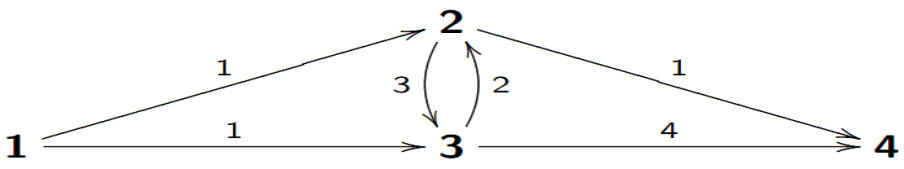
La dernière étape est utile pour calculer une solution optimale, et pas seulement la valeur optimale.
Les 3 premières étapes sont impératives pour la résolution d'un problème par la méthode de programmation dynamique. On peut omettre l'étape 4 si seule la valeur d'une solution optimale nous incombe. Lorsqu'on effectue l'étape 4, on gère parfois des informations supplémentaires pendant le calcul de l'étape 3 pour faciliter la construction d'une solution optimale.

## 3. QUAND UTILISER LA PROGRAMMATION DYNAMIQUE ?

On peut bien se demander dans quelle situation envisager une solution à base de programmation dynamique. Dans cette mini section, nous allons examiner les deux grandes caractéristiques que doit posséder un problème d'optimisation pour que la programmation dynamique soit applicable : **sous-structure optimale** et **chevauchement des sous-problèmes**.

### ➢ Sous-Structure optimale

La première étape de la résolution d'un problème d'optimisation via la programmation dynamique est de caractériser la structure d'une solution optimale. Retenons qu'un problème fait apparaitre une **sous-structure optimale** si une solution optimale au problème contient en elle des solutions optimales aux sous-problèmes. La présence d'une sous-structure optimale est un bon indice de l'utilité de la programmation dynamique (mais cela peut aussi signifier qu'une stratégie *gloutonne* est applicable). Avec la programmation dynamique, on construit une solution optimale du problème à partir de solutions optimales de sous-problèmes. Par conséquent, on doit penser à vérifier que la gamme des sous-problèmes que l'on considère inclut les sous-problèmes utilisés dans une solution optimale ; c'est le *principe d'optimalité de Bellman*.
Bien que naturel, le principe n'est pas toujours applicable ! Prenons l'exemple de la recherche du plus long chemin élémentaire (sans cycle, ni boucle) d'un point à un autre dans un graphe. Si le chemin élémentaire le plus long de A à C passe par B, le tronçon de A à B de ce chemin n'est pas forcément le plus long chemin élémentaire de A à B! Pourquoi ? Considérons le graphe suivant :




Le plus long chemin simple de **1** à **4** est **1⟶2⟶3⟶4**.

**Mais,** le plus long chemin simple de **1** à **2** : **1⟶3⟶2** ; qui n'est pas un sous-chemin du chemin optimal. D'où le Principe d'optimalité de Bellman non satisfait, donc programmation dynamique pas utilisable ici.

Cet exemple montre que, pour les plus longs chemins élémentaires, non seulement il manque une sous-structure optimale, mais en plus on ne peut pas toujours construire une solution « licite » à partir de solutions de sous-problèmes. Si l'on combine les plus longs chemins élémentaires : **1⟶3⟶2** et **2⟶3⟶4**, on obtient le chemin **1⟶3⟶2⟶3⟶4** qui n'est pas élémentaire. En fait, le problème consistant à trouver un plus long chemin élémentaire ne semble pas avoir de sous-structure optimale de quelque sorte que ce soit. Aucun algorithme efficace de programmation dynamique n'a pu être trouvé pour ce problème. En fait, ce problème est **NP-complet** (voir l'exposé sur la NP-Complétude), ce qui implique qu'il est peu probable qu'il puisse être résolu en temps polynomial.

- **Chevauchement des sous problèmes**

La seconde caractéristique que doit avoir un problème d'optimisation pour que la programmation dynamique lui soit applicable est la suivante : l'espace des sous-problèmes doit être « réduit », au sens où un algorithme récursif pour le problème résout constamment les mêmes sous-problèmes au lieu d'en engendrer toujours de nouveaux. En général, le nombre total de sous-problèmes distincts est polynomial par rapport à la taille de l'entrée. Quand un algorithme récursif repasse sur le même problème constamment, on dit que le problème d'optimisation contient des sous-problèmes qui se chevauchent. A contrario, un problème pour lequel l'approche diviser-pour-régner est plus adaptée génère, le plus souvent, des problèmes nouveaux à chaque étape de la récursivité.

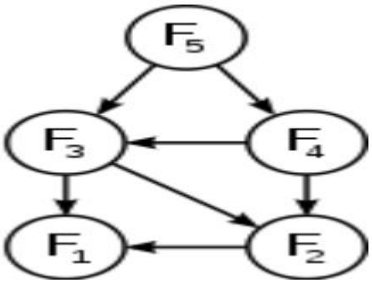



**Figure 1** : Le graphe de dépendance des sous-problèmes pour calculer le terme de rang 5 de la suite de Fibonacci sv. Le calcul de F5 et F4 se chevauchent en F3, F2 et F1.

# II. ETUDE DE QUELQUE EXEMPLES D'APPLICATION DE LA PROGRAMMATION DYNAMIQUE

## 1. PROBLEME DU VOYAGEUR DE COMMERCE

Dans cet exemple nous allons utiliser la programmation dynamique pour résoudre le problème du voyageur de commerce.

**Problème réel :** le problème est "étant donné un ensemble de ville à visiter, dans quel ordre doit-on les parcourir pour minimiser le coût".

**Modélisation:** modélisons notre problème comme suit:

Soit **N**={1,2,…,n} l'ensemble des villes à visiter, **D [i, S]** la distance d'un plus court chemin partant de i, passant par tous les points de S, une et une seule fois, et se terminant au sommet 1. La relation suivante n'est donc pas difficile à établir :

$$D\,[i, S] = \min_{j \in S}\{d_{ij} + D[j, S - \{\, j \,\}\,]\}$$

La distance entre deux sommets i et j est notée par $d_{ij}$.

Pour résoudre ce problème, on proposera deux approches:

**Approche 1:** il s'agit d'une approche naïve qui consiste à calculer toute les possibilités que l'on puisse avoir avec toutes les villes à notre disposition.

**Approche 2:** il s'agit ici de résoudre ce problème par la méthode de la programmation dynamique:

### 1. Caractériser la structure d'une solution optimale

Il s'agit de montrer que la propriété de sous-structure optimale est respectée par le problème du voyageur de commerce ; C'est-à-dire qu'une solution optimale pour un problème amène à la solution optimale pour tous les sous problèmes.

Soit **C** l'itinéraire de coût minimal passant par les points de 1 à k puis de k+1 à 1**,** le principe de sous structure optimale stipule que cet itinéraire est optimal si le chemin quittant de 1 à k est optimal, ainsi que celui de k+1 à 1. Supposons qu'il existe un autre chemin optimal quittant de k+1 a 1 ; remplacer ce chemin à l'intérieure du cycle hamiltonien optimal produirait un autre cycle hamiltonien de poids inférieur à l'optimum, d'où l' absurdité. Ceci montre que le problème du voyageur de commerce possède bien la propriété de sous structure optimale.

## 2. Définir récursivement la valeur d'une solution optimale

Étant donnée 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 une 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 *dij*.
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}.

Il existe deux grandes catégories de méthodes de résolution : les méthodes exactes et les méthodes approchées. Les méthodes exactes permettent d'obtenir une solution optimale à chaque fois, mais le temps de calcul peut être long si le problème est compliqué à résoudre. Les méthodes approchées, encore appelées heuristiques, permettent quant à elles d'obtenir rapidement une solution approchée, mais qui n'est donc pas toujours optimale.
Ainsi, dans le jargon informatique il existe plusieurs méthodes exactes de résolution du problème de voyageur de commerce, mais nous allons utiliser celle en corrélation avec la **programmation dynamique** :
Étant donné **D [i, S]** la distance d'un plus court chemin partant de i, passant par tous les points de S, une et une seule fois, et se terminant au sommet 1 ; Et sachant que

$$D\,[i, S] = \min_{j \in S}\{d_{ij} + D[j, S - \{\,j\,\}\,]\}$$

, avec $d_{ij}$ la distance entre deux sommets i et j.
La solution optimale est donc donnée par **D [1, V- {1}]**, tel que :

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

Il est clair que $\mathbf{D\,[i, \emptyset] = d_{i1}}$.
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 $2^{n-1} - 1$.

## 3. Calculer la valeur d'une solution optimale

Supposons n = 4, on aura à construire la table suivante :

| | {} | {2} | {3} | {4} | {2,3} | {2,4} | {3,4} |
|---|---|---|---|---|---|---|---|
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |

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, Ø] = $d_{21}$ = 5 ;
D [3, Ø] = $d_{31}$ = 6 ;
D [4, Ø] = $d_{41}$ = 8 ;
D [2,{3}] = $d_{23}$ + D [3, Ø] = 15 ;
D [2,{4}] = 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 ≠1 ; i ∉ 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.

### 4. Construction d'une solution optimale à partir des informations calculées

Pour le parcours à faire ayant la valeur optimale de 35, 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 commence donc 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.**

**COMPLEXITE TEMPORELLE**

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 ($n^2$ $2^n$)**

## 2. PROBLEME DU SAC A DOS

# PROGRAMMATION DYNAMIQUE

Notre deuxième exemple de programmation dynamique est un algorithme qui résout le problème du sac à dos.

**Problème réel** : L'énoncé de ce fameux problème est simple : « Étant donné plusieurs objets possédants chacun un poids et une valeur et étant donné un poids maximum pour le sac, quels objets faut-il mettre à l'intérieur du sac de manière à maximiser la valeur totale sans dépasser le poids maximal autorisé pour le sac ? »

**Modélisation** : L'énoncé du problème étant assez simple à comprendre, la modélisation ne dérogera pas à la règle. Celle qu'on a choisi sera énoncée telle quelle :
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 $\mathbf{W_i > 0}$ et un gain $\mathbf{V_i > 0}$. 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 au cours de cette section, un objet est soit choisi soit ignoré. Autrement dit, les objets sont indivisibles, et de ce fait nous ne pouvons prendre une portion d'un objet dans le sac.
En termes mathématiques, nous avons ce qui suit :

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

$$\sum w_i x_i \leq W$$

$$x_i = 0, 1.$$

Pour résoudre ce problème afin d'obtenir une solution optimale, l'on proposera deux solutions :
**Idée 1 : méthode brute :** L'approche naïve pour gérer ce problème serait celle de générer l'ensemble de toutes les combinaisons possibles que l'on peut avoir avec les objets à notre disposition sans tenir compte d'une quelconque contrainte. Ensuite on vérifiera parmi ces combinaisons celle qui satisfassent la contrainte de poids. Pour finir on comparera le gain généré par chacune de ces combinaisons afin de trouver la solution optimale.

## 1- Algorithme

```
69
70  int probleme_sac_a_dos(int *tabPoids, int *tabGain, int nbObjets, int poidsMax)
71  {
72      int maxGain = 0;
73      for(int i=0;i < pow(2,nbObjets); i++){
74          int *combination = decimalToBinaryBaseN(i);
75          int poids = calculPoids(combination,tabPoids);
76          int gain = calculGain(combination,tabGain);
77          if(poids <= poidsMax && gain > maxGain) {
78              maxGain = gain;
79              printf("The gain of %d combination equals to %d is greater than max gain %d. \n",i,gain,maxGain);
80          }
81      }
82      printf("\n--->Finally the maximum gain is : %d",maxGain);
83      return maxGain;
84  }
85
```

### 2- Complexité en temps

De façon naïve on aura pour complexité :

$$T(n) = 2^n (n + n + n)$$
$$= O(2^n n)$$

Cet algorithme a donc une complexité exponentielle en temps. Et bien qu'en est-il de sa complexité en espace ?

### 3- Complexité en espace

L'exécution de cet algorithme met en œuvre, en premier plan l'utilisation un tableau de dimension $2^n$ (nombre d'objet) dans lequel sont stockées toutes les combinaisons possibles d'objets. On note également l'utilisation de deux tableaux de taille n pour stocker l'ensemble des poids et gains de chaque objet. Ainsi l'espace alloué par cet algorithme pour s'exécuter en mémoire est de l'ordre de grandeur de la taille de ce tableau. D'où :

$$T(n) = O(2^n)$$

**Idée 2 : Programmation dynamique :** Il s'agit ici de résoudre ce problème à l'aide de la méthode de programmation dynamique.

### 1. Caractériser la structure d'une solution optimale

Soit $X_1X_2X_3...X_{n-1}X_n$ une séquence de n objets constituant une solution optimale pour notre problème du sac à dos.

Admettre que le problème du sac à dos possède la propriété de sous structure optimale revient à admettre que la sous séquence $X_1X_2X_3...X_{n-1}$ se veut elle aussi optimale. Sachant que l'optimalité est également fonction du poids, cette sous séquence doit être optimale pour un poids donné.

Raisonnons par l'absurde et supposons que l'on peut trouver une sous séquence plus optimale que $X_1X_2X_3...X_{n-1}$. Vu qu'une sous séquence est également fonction d'un poids, l'on est amené à effectuer une disjonction de cas :

➔ **Si $X_n = 0$**

Alors $X_1X_2X_3...X_{n-1}0$ est la séquence optimale de notre problème du sac à dos dans $W$. Supposons que l'on ait une sous séquence $Y_1Y_2...Y_{n-1}$ plus optimale que $X_1X_2X_3...X_{n-1}$ dans W. En ajoutant la variable décisionnelle associée à l'objet n, l'on obtiendra la séquence $Y_1Y_2...Y_{n-1}0$ qui est plus optimale que $X_1X_2X_3...X_{n-1}0$ dans $W$ **d'où l'absurdité.**

➔ **Si $X_n = 1$**

Alors $X_1X_2X_3...X_{n-1}1$ est la séquence optimale de notre problème du sac à dos dans $W$. Supposons que l'on ait une sous séquence $Y_1Y_2...Y_{n-1}$ plus optimale que $X_1X_2X_3...X_{n-1}$ dans $W - W_n$. En ajoutant la variable décisionnelle associée à l'objet n, l'on obtiendra la séquence $Y_1Y_2...Y_{n-1}1$ qui est plus optimale que $X_1X_2X_3...X_{n-1}1$ dans $W$ **d'où l'absurdité.**

A travers cette disjonction, l'on conclue qu'une séquence optimale contient en elle des sous séquences optimales et par conséquent que notre problème du sac à dos possède la propriété de sous structure optimale.

**2. Définir récursivement la valeur d'une solution optimale**

Pour pouvoir définir récursivement ce problème il serait préférable de le diviser en deux problèmes 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 *j,* alors résoudre le problème revient à trouver la valeur de $P_{n,W}$. En calculant $P_{i,j}$ la séquence d'objets peut être divisée en deux :

- **Les (*i*-1) premiers objets.**
- **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 *vi*. 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 donc : $P_{i,j} = 0$ pour $i = 0$ ou $j = 0$. Cela nous amène aux relations récursives suivantes :

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

### 3. Calculer la valeur d'une solution optimale

Pour calculer la valeur de la solution optimale on vous propose cet algorithme qui résulte essentiellement de la traduction de la formule récursive trouvée au titre précèdent.

```
4   int probleme_sac_a_dos_with_dp(int *tabPoids, int *tabGain, int nbObjets, int poidsMax) {
5       int P[nbObjets + 1][poidsMax + 1];
6           // INITIALISATION DU TABLEAU
7       for(int i=0; i<=nbObjets; i++)
8           P[i][0] = 0;
9       for(int i=0; i<=poidsMax ;i++)
10          P[0][i] = 0;
11
12      for(int i=1; i<=nbObjets; i++){
13          for(int j=1;j<=poidsMax; j++){
14              P[i][j] = P[i-1][j];
15              if (j >= tabPoids[i-1] ){
16                  if( ((P[i-1][j-tabPoids[i-1]]) + tabGain[i-1]) > P[i-1][j] ){
17                          P[i][j] = P[i-1][(j-tabPoids[i-1])] + tabGain[i-1];
18
19      }}}}
20      printf("\n--->Finally the maximum gain is : %d", P[nbObjets][poidsMax]);
21      return P[nbObjets][poidsMax];
22  }
23
24
```

***) Complexité en temps**

La complexité en temps de la méthode récursive est la suivante :

$$T(n) = n + k + (k*n)$$
$$= O(k*n)$$

Avec k le poids que le sac peut supporter et n le nombre d'objets dont on dispose.

***) Complexité en espace**

L'exécution de cet algorithme met en œuvre l'utilisation un tableau de dimension n (nombre d'objet) * w (capacité maximale du sac). Ainsi lorsque le jeu de données en entrée devient très important, la complexité en espace de cet algorithme correspond à la taille du tableau recensant les Pi, j calculé par l'algorithme. Ainsi l'espace alloué par cet algorithme pour s'exécuter en mémoire est de l'ordre de grandeur de la taille de ce tableau ; d'où :

$$T(n) = O(n*w) ;$$

### 4. Construction d'une solution optimale à partir des informations calculées

Malgré le fait que l'algorithme proposé ci-dessus nous permette de calculer le gain maximal des n objets, il ne nous permet pas de déterminer quels sont les objets qui contribuent à l'obtention de ce gain. Cela nous permet donc de comprendre que le travail n'est pas achevé car c'est ce choix d'objets qui constitue la solution au problème réel. Comment parvenir donc pour pouvoir construire cette solution optimale ?

Et bien il nous suffit tout simplement de faire un simple backtrack (effectuer le chemin arrière) de la solution optimale et de déterminer quel choix nous a permis d'aboutir à cette solution. Sachant que :

$$P_{i,j} = \max \{P_{i-1,j} ; P_{i-1,j-wi} + v_i\}$$

Si c'est le premier terme qui nous a permis d'attendre le gain max, on déduit que l'objet i a été ignoré et on backtrack $P_{i-1,j}$. Au cas contraire l'on déduit qu'il a été mis dans le sac et on backtracke le second terme. L'on s'arrête une fois qu'on obtient un résultat impossible à backtracker tel que P[X,0].

## 3. PRODUITS MATRICIELS ENCHAINES

Notre troisième exemple de programmation dynamique est un algorithme qui résout le problème des produits matriciels enchaînés.

**Problème réel** : Soit n matrices $\mathbf{M}_1$, $\mathbf{M}_2$, ..., $\mathbf{M}_n$ à multiplier, on souhaite calculer le produit $\mathbf{M}_1\mathbf{M}_2...\mathbf{M}_n$.

**Prérequis** :

- Pour faire le produit $\mathbf{M}_1\mathbf{M}_2$, il est nécessaire que le nombre de colonnes de **M**1 soit égal au nombre de ligne de $\mathbf{M}_2$
- Le nombre de multiplications scalaires engendrées par le produit $\mathbf{M_1M_2}$**,** de dimensions respectives (**p x q**) et (**q x r**) est égal à **p x q x r.**
- La multiplication matricielle est une opération associative **(**$M_1(M_2M_3) = (M_1M_2)\ M_3 = M_1M_2M_3$).

**Modélisation** : Étant donné que la multiplication matricielle est une opération associative, il existe une multitude de manières d'effectuer le produit entre les matrices. Chacune des manières correspond à un parenthésage unique du produit qui lève l'ambiguïté sur l'ordre de multiplication des matrices et qui conduit au même résultat. Un produit de matrices **entièrement parenthésé** est soit une matrice unique, soit le produit de deux produits matriciels entièrement parenthésés.

Ainsi, si nous considérons les matrices $M_1$, $M_2$, $M_3$, $M_4$, *le produit* $M_1M_2M_3\ M_4$ *peut être entièrement parenthésés de 5 façons différentes (Donc être calculé de 5 façon différentes) à savoir :*

($M_1$($M_2$ ($M_3$ $M_4$))),
($M_1$ (($M_2$ $M_3$) $M_4$)),
(($M_1$ $M_2$)( $M_3$ $M_4$)),
(($M_1$ ($M_2$ $M_3$)) $M_4$),
((($M_1$ $M_2$) $M_3$) $M_4$).

La manière dont un produit de matrices est entièrement parenthésé peut avoir un impact critique sur le nombre de multiplication scalaire nécessaire à son calcul. Par exemple, soit les 4 matrices suivantes :

**A** ; **B** ; **C**; **D** de dimensions respectives : 13×5 ; 5×89 ; 89×3 ; 3×34. Si on calcule le produit ABCD selon le parenthésage :

- (((*AB*)*C*)*D)* : *le coût d'évaluation du produit en termes de multiplication scalaire sera de* 10582 multiplications scalaires.
- ((*AB*)(*CD*)) : *le coût d'évaluation du produit en termes de multiplication scalaire sera de* 52201 multiplications scalaires.
- ((*A*(*BC*))*D)* : *le coût d'évaluation du produit en termes de multiplication scalaire sera de* 2856 multiplications scalaires.
- *(A*((*BC*)*D*)) : *le coût d'évaluation du produit en termes de multiplication scalaire sera* 4055 multiplications scalaires.
- *(A*(*B*(*CD*))) : *le coût d'évaluation du produit en termes de multiplication scalaire sera* 26 418 multiplications scalaires.

Donc pour cet exemple la meilleure manière de calculer le produit ABCD est de le faire suivant le parenthésage ((*A*(*BC*))*D).*

*Au vu de tous cela, se pose la question de savoir comment trouver une meilleure parenthétisation pour un calcul de produits matriciels ? autrement dit le problème des* produits matricielles enchainés peut être énoncer comme suit : Soit n matrices $\mathbf{M}_1$, $\mathbf{M}_2$, ..., $\mathbf{M}_n$, où pour $i = 1, 2, \ldots, n$, la matrice $M_i$ a la dimension $p_{i-1} \times p_i$, parenthéser entièrement le produit $\mathbf{M}_1\mathbf{M}_2...\mathbf{M}_n$ de façon à minimiser le nombre de multiplications scalaires. Par conséquent, dans le problème des multiplications matricielles enchaînées, on ne multiplie pas vraiment les matrices. On cherche plutôt un ordre de multiplication qui minimise le coût de l'évaluation du produit en termes de nombre de multiplication scalaire.
Pour répondre à cette question deux idées sont à examiner :

- **Idée 1 : méthode brute** : il s'agit ici d'essayer toutes les possibilités. En effet on insère les parenthèses de toutes les manières possibles et ensuite, pour chacune d'elle, on compte le nombre de multiplications scalaires engendrées.
  Pour ce faire procédons comme pour diviser et régner en subdivisant le problème en deux sous-problèmes comme suit :

  $M = (M_1M_2...M_i)\,(M_{i+1}M_{i+2}...M_n)$
  Si nous notons par **P**(**n**) le nombre de parenthésage possible du produit $\mathbf{M}_1\mathbf{M}_2...\mathbf{M}_n$ (c'est-à-dire le nombre de parenthésage possible d'un produit de n matrices), alors **P**(**i**) va représenter le nombre de parenthésage possible du produit $(M_1M_2...M_i)$ et **P** (**n** - **i**) celui de $(M_{i+1}M_{i+2}...M_n)$. Quand **n** = 1, il n'y a qu'une seule matrice et donc une seule façon de parenthéser entièrement le produit. Quand **n** ≥ 2, un produit matriciel entièrement parenthésé est le produit de deux sous-produits matriciels entièrement parenthésés, et la démarcation entre les deux sous-produits peut intervenir entre les **i**ème et (**i** + 1) ème matrices, pour tout **i** = 1, 2, . . ., **n** - 1. On obtient donc la récurrence :

$$P(n) = \begin{cases} 1 & \text{si } n = 1\,, \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geqslant 2\,. \end{cases}$$

**Résolvons cette équation de récurrence :**

On a : $P(n) = \Omega(2^n)$.
P(n) est donc au moins exponentiel en **n**, de ce fait la méthode brute est moins efficace pour déterminer un parenthésage optimal de produits matriciels enchaînés.

- **Idée 2 : Programmation dynamique :** il s'agit ici de résoudre ce problème à l'aide de la méthode de programmation dynamique.

### 1. Caractériser la structure d'une solution optimale

Il s'agit ici de vérifier que le problème des produits matriciels enchaînés possède la propriété de sous-structure optimale.

Nous noterons par $\mathbf{M_{i..j}}$, avec $\mathbf{i \leq j}$, la matrice résultante de l'évaluation du produit $\mathbf{M_iM_{i+1} \cdots M_j}$. Pour $\mathbf{i < } j$, tout parenthésage optimal du produit $\mathbf{M_iM_{i+1} \cdots M_j}$ sépare le produit entre $\mathbf{M_k}$ et $\mathbf{M_{k+1}}$ pour un certain **k** de l'intervalle $\mathbf{i \leq } k < \mathbf{j}$. Autrement dit, pour une certaine valeur de **k**, on commence par calculer les matrices $\mathbf{M_{i..k}}$ et $\mathbf{M_{k+1..j}}$, puis on les multiplie ensemble pour obtenir le résultat final $\mathbf{M_{i..j}}$. Le coût de ce parenthésage optimal est donc le coût du calcul de la matrice $\mathbf{M_{i..k}}$, plus celui du calcul de $\mathbf{M_{k+1..j}}$, plus celui de la multiplication de ces deux matrices. Supposons qu'un parenthésage optimal de $\mathbf{M_iM_{i+1} \cdots M_j}$ fractionne le produit entre $\mathbf{M_k}$ et $\mathbf{M_{k+1}}$. Alors, le parenthésage du sous-produit $\mathbf{M_iM_{i+1} \cdots M_k}$ à l'intérieur du parenthésage optimal de $\mathbf{M_iM_{i+1} \cdots M_j}$ est forcément un parenthésage optimal de $\mathbf{M_iM_{i+1} \cdots M_k}$. En effet s'il existait un parenthésage meilleure de $\mathbf{M_iM_{i+1} \cdots M_k}$, remplacer ce parenthésage dans le parenthésage optimal de $\mathbf{M_iM_{i+1} \cdots M_j}$ produirait un autre parenthésage de $\mathbf{M_iM_{i+1} \cdots M_j}$ dont le coût serait inférieur au coût du premier parenthésage optimale de $\mathbf{M_iM_{i+1} \cdots M_j}$ que l'on a considéré :
On arrive à une contradiction. On peut faire la même observation pour le parenthésage du sous-produit $\mathbf{M_{k+1}M_{k+2} \cdots M_j}$ à l'intérieur du parenthésage optimal de $\mathbf{M_iM_{i+1} \cdots M_j}$ : c'est forcément un parenthésage optimal de $\mathbf{M_{k+1}M_{k+2} \cdots M_j}$.

### 2. Définir récursivement la valeur d'une solution optimale

Comme le titre l'indique, il s'agit ici de définir récursivement la valeur d'une solution optimale en fonction des solutions des sous-problèmes.

Pour le problème des multiplications matricielles enchaînées, on prend comme sous-problèmes les problèmes consistant à déterminer le coût minimum de parenthésages de $\mathbf{M_iM_{i+1} \cdots M_j}$ pour $1 \leq \mathbf{i \leq } j \leq \mathbf{n}$. Soit **m** [**i**,**j**] le nombre minimum de multiplications scalaires nécessaires pour le calcul de la matrice $\mathbf{M_{i..j}}$ ; pour le problème entier, le coût d'un parenthésage optimale pour calculer $\mathbf{M_{1..n}}$ sera donc **m** [1, **n**].

On a donc pour :

i = j : aucune multiplication : **m [i,j]** = 0

- i < j : **m [i,j]** = Coût minimum pour calculer la matrice $\mathbf{M_{i..k}}$
  +
  Coût minimum pour calculer la matrice $\mathbf{M_{k+1...j}}$
  +
  Coût pour multiplier les matrices $\mathbf{M_{i..k}}$ et $\mathbf{M_{k+1..j}}$ **(qui vaut** $p_{i-1} \times p_k \times p_j$**).**

**C'est-à-dire : m[i,j]** = m*[*i,k*] + m*[*k + 1**,** j*] + $p_{i-1} \times p_k$ **x** $p_j$
Cette équation récursive suppose que l'on connaisse la valeur de **k**, ce qui n'est pas le cas. Tous ce que l'on sait est qu'il existe **j** - **i** valeurs possibles pour **k**, à savoir **k** = **i**, **i** + 1, . . ., **j** – 1 et étant donné qu'un parenthésage optimal doit utiliser l'une de ces valeurs pour **k**, il nous suffit de toutes les vérifier pour trouver la meilleure. Par conséquent le coût minimum de parenthésage de $\mathbf{M_i M_{i+1}} \cdots \mathbf{M_j}$ est *définit récursivement de la manière suivante :*

$$m[i,j] = \begin{cases} 0 & \text{si } i = j \\ \min\limits_{i \leqslant k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{si } i < j \end{cases}$$

Les valeurs **m** [**i**,**j**] donnent les coûts des solutions optimales des sous-problèmes.

### 3. *Calculer la valeur d'une solution optimale*

*Il s'agit ici de faire ce calcul en utilisant l'approche ascendante de la programmation dynamique.*

On constate que, le nombre de sous-problèmes est assez réduit : un problème pour chaque choix de $i$ et de $j$ tels que$1 \leq i \leq j \leq n$, soit au total $C_n^2 + n = \Theta(n^2)$ . Un algorithme récursif peut alors rencontrer chaque sous-problème plusieurs fois dans différentes branches de son arbre de récursivité. Autrement dit le problème des produits matriciels enchainés contient des sous-problèmes qui se chevauchent.
L'algorithme *ORDRE-CHAÎNE-MATRICES* ci-dessous utilise un tableau auxiliaire **m**[1 . . **n**,1 . . **n**] pour stocker les coûts minimums **m**[**i**, **j**] et un tableau auxiliaire **s**[1 . . **n**,1 . . **n**] qui mémorise quel est l'indice de **k** qui avait donné le coût minimum lors du calcul de **m**[**i**,**j**]. L'entrée de cet algorithme est une séquence ($p_0$, $p_1$, . . ., $p_n$), où *longueur*[*p*] = *n* + 1.3

**Pseudo code :**

```
ORDRE-CHAÎNE-MATRICES(p)
 1  n ← longueur[p] − 1
 2  pour i ← 1 à n
 3      faire m[i, i] ← 0
 4  pour l ← 2 à n                ▷ l est la longueur de la chaîne.
 5      faire pour i ← 1 à n − l + 1
 6              faire j ← i + l − 1
 7                   m[i, j] ← ∞
 8                   pour k ← i à j − 1
 9                       faire q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                            si q < m[i, j]
11                              alors m[i, j] ← q
12                                    s[i, j] ← k
13  retourner m et s
```

L'algorithme commence par l'affectation **m** [**i**, **i**] ← 0, pour **i** = 1, 2, . . ., **n** (coûts minimums pour les chaînes de longueur 1) aux lignes 2–3. Il utilise ensuite la récurrence définie à l'étape 2 pour calculer **m** [**i**, **i**+1] pour **i** = 1, 2, . . ., **n**-1 (coûts minimums pour les chaînes de longueur

2) pendant la première exécution de la boucle des lignes 4–12. Au deuxième passage dans la boucle, il calcule **m** [**i**, **i** + 2] pour **i** = 1, 2, . . ., **n** – 2 (coûts minimums pour les chaînes de longueur 3), et ainsi de suite. A chaque étape, le coût **m** [**i**, **j**] calculé aux lignes 9–12 ne dépend que des éléments de tableau **m** [**i**, **k**] et **m** [**k** + 1, **j**] déjà calculés.

**Complexité :**

On a T(n) = Θ ($n^3$). ORDRE-CHAÎNE-MATRICES est donc beaucoup plus efficace que la méthode brute.

**Exemple :**

*On considère les matrices* $M_1, M_2, M_3, M_4, M_5$ et $M_6$ de dimensions respectives *:* $30 \times 35; 35 \times 15; 15 \times 5; 5 \times 10; 10 \times 20$ et $20 \times 25$.
*L'entrée de l'algorithme est alors la séquence (30, 35,15, 5, 10, 20,25) et n = 6*
La figure 1 donne un aperçu des tableaux m et s après exécutions de l'algorithme *ORDRE-CHAÎNE-MATRICES* sur l'entrée *(30, 35,15, 5, 10, 20, 25)*. Comme nous n'avons défini **m** [**i**, **j**] que pour **i ≤ j**, seule la partie du tableau **m** strictement supérieure à la diagonale principale est utilisée. La figure présente le tableau de façon à faire apparaître la diagonale principale horizontalement. La chaîne de matrices est donnée en bas. Chaque ligne horizontale du tableau contient les éléments pour les chaînes de matrices de même longueur. ORDRE-CHAÎNE-MATRICES calcule les lignes du bas vers le haut, et de gauche à droite à l'intérieur de chaque ligne.

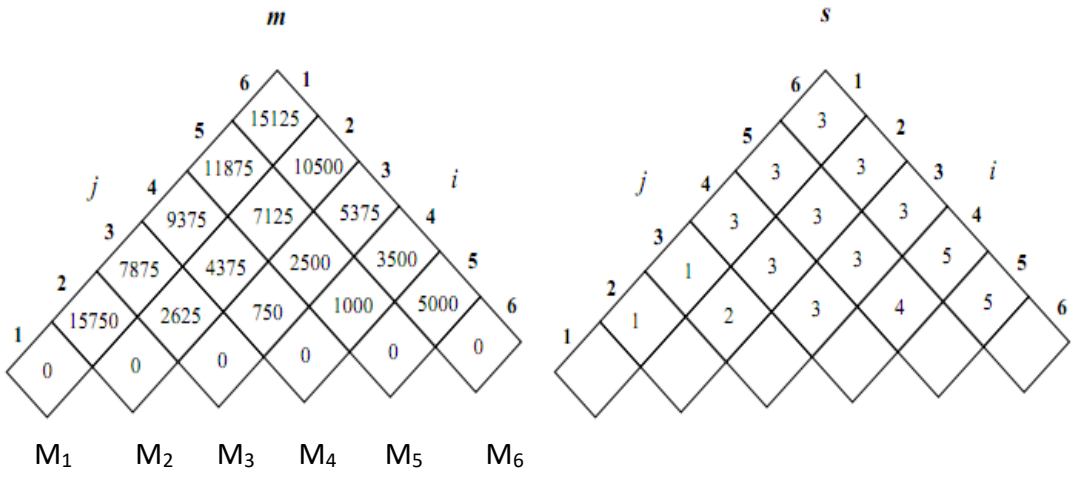



Figure 1 : Tableaux m et s calculés par ORDRE-CHAÎNE-MATRICES pour $n = 6$ et les dimensions des matrices $M_1, M_2, M_3, M_4, M_5$ et $M_6$ *sont* respectivement : $30 \times 35; 35 \times 15; 15 \times 5; 5 \times 10; 10 \times 20$ et $20 \times 25$.

### 4. Construction d'une solution optimale à partir des informations calculées

Bien qu'ORDRE-CHAÎNE-MATRICES détermine le nombre minimum de multiplications scalaires nécessaires pour calculer le produit d'une suite de matrices, elle ne montre pas directement comment multiplier les matrices. Toutefois Il n'est pas difficile de construire une solution optimale à partir des données calculées et mémorisées dans le tableau **s** [1...**n**, 1...**n**]. La procédure récursive AFFICHE-PARENTHÉSAGE-OPTIMAL affiche un parenthésage optimal

de $\mathbf{M_iM_{i+1}} \cdot \cdot \cdot \mathbf{M_j}$, à partir du tableau *s* calculé par ORDRE-CHAÎNE-MATRICES. AFFICHE-PARENTHÉSAGE-OPTIMAL prend alors 3 paramètres en entrée à savoir : s, i et j.

**Pseudo code :**

```
AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, i, j)
1  si i = j
2        alors afficher« M »i
3        sinon afficher« ( «
4              AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, i, s[i, j])
5              AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, s[i, j] + 1, j)
6              print » ) »
```

Dans l'exemple de la figure 1, l'appel AFFICHAGE-PARENTHÉSAGE-OPTIMAL (**s**, 1, 6) affiche le parenthésage ((**A**1(**A**2**A**3)) ((**A**4**A**5) **A6**)).
Il vient que l'algorithme **AFFICHE**-PARENTHÉSAGE-OPTIMAL permet de trouver un parenthésage optimal pour un calcul de produits matriciels enchainés. Elle résout donc le problème d'optimisation cependant elle ne résout pas le problème réel. Toutefois pour résoudre le problème réel on peut écrire l'algorithme MULTIPLICATION-CHAÎNE-MATRICES (*M*, *s*, *i*, *j*) qui effectue les multiplications matricielles enchaînées proprement dites, et ce à partir des matrices $\mathbf{M_iM_{i+1}} \cdot \cdot \cdot \mathbf{M_j}$, du tableau *s* calculé par ORDRE-CHAÎNE-MATRICES, et des indices *i* et *j*.

**Pseudo code :**

```
MULTIPLICATION-CHAÎNE-MATRICES (M, s, i, j)
1    si i = j
2      alors
3        renvoyer Mi
4    sinon
5        X ← MULTIPLICATION-CHAÎNE-MATRICES (M, s, i, s[i,j])
6        Y ← MULTIPLICATION-CHAÎNE-MATRICES (M, s, s[i,j]+ 1, j)
7        Renvoyer MULTIPLIER-MATRICES (X, Y)
```

**NB** : MULTIPLIER-MATRICES **est une fonction qui multiplie deux matrices passées en paramètres.**

## III. COMPARAISON DE LA PROGRAMMATION DYNAMIQUE ET D'AUTRES METHODES ALGORITHMIQUES

# 1. METHODE DE PROGRAMMATION DYNAMIQUE ET METHODE DU DIVISER POUR REGNER

La méthode de programmation dynamique, comme la méthode DPR, résout des problèmes en combinant des solutions de sous-problèmes.
Les algorithmes diviser-pour-régner partitionnent le problème en sous-problèmes indépendants qu'ils résolvent récursivement, puis combinent leurs solutions pour résoudre le problème initial. Par conséquent la méthode diviser-pour-régner est inefficace si on doit résoudre **plusieurs fois le même sous-problème**. Donc :

- Avec la méthode DPR le graphe de dépendance des sous-problèmes pour un problème donné est un arbre.
- Avec la méthode de programmation dynamique le graphe de dépendance des sous-problèmes pour un problème donné n'est pas un arbre : cela montre clairement que les sous-problèmes se chevauchent (se superposent). Par conséquent on peut alors dire que La programmation dynamique consiste à stocker les valeurs des sous-problèmes pour éviter les recalcules.

# 2. METHODE DE PROGRAMMATION DYNAMIQUE ET METHODE GLOUTONNE

Les techniques de programmation dynamique et de choix glouton reposent toutes deux sur **l'utilisation de sous-problèmes induits par le problème initial.**
Il y a cependant une grosse différence entre elles. En programmation dynamique, **on calcule les solutions de tous les sous-problèmes** et on les combine de façon optimale. Tandis qu'avec une méthode gloutonne **on choisit à tout moment ce qui semble être la meilleure solution** et on résout le sous problème qui en découle.
La comparaison des arbres, des appels récursifs montre bien cette différence. En programmation dynamique **on explore toutes les branches une et une seule fois**, alors que dans une méthode gloutonne **on explore une seule branche**.
Chacune de ces deux techniques possèdent ces avantages et inconvénients, En particulier, un algorithme glouton sera clairement de **complexité moindre** qu'un algorithme en programmation dynamique mais en contrepartie la programmation dynamique **fournit toujours les solutions optimales** ce qui n'est pas le cas avec l'approche gloutonne.
**NB** : Un algorithme glouton peut toujours être convertir en un algorithme de programmation dynamique. Même si c'est parfois délicat. Mais l'inverse est faux.

# 3. METHODE DE PROGRAMMATION DYNAMIQUE ET METHODE DE PROGRAMMATION LINEAIRE

Comme la programmation dynamique, la programmation linéaire concerne la maximisation ou la minimisation d'un système.

- La programmation linéaire est généralement présentée sous forme de fonction linéaire (Max $x_1+x_2$), accompagné d'un ensemble de contrainte sous forme d'équations ou d'inéquations ($x_1+2x_2>5$) ; il est donc question d'optimiser la fonction en respectant ces contraintes. La programmation dynamique quant à elle optimise le système en suivant un processus de décision séquentielle.

- La programmation linéaire consiste à visiter les sommets d'un ensemble convexe de façon à améliorer progressivement la valeur de la fonction alors que La programmation dynamique, elle, consiste à résoudre les problèmes élémentaires puis de plus en plus grand afin de ressortir la solution optimale.

- La notion de sous problèmes n'intervient pas en programmation linéaire alors que c'est le fondement de la programmation dynamique.

- La programmation linéaire peu se résoudre graphiquement contrairement à la programmation dynamique.

# IV. LA PROGRAMMATION DYNAMIQUE DANS LES DOMAINES DE RECHERCHES

Le cycle Master étant celui de l'initiation à la recherche, l'on a pensé qu'il était juste et judicieux de notre part de parler brièvement de l'apport de la programmation dynamique dans divers domaines de la recherche scientifique afin de rappeler et montrer une fois de plus le rôle capital qu'elle peut jouer dans l'optimisation d'un problème donné. C'est ainsi que l'on observe ses apports sur divers domaines tel que celui de l'environnement, celui de la biologie ou encore des transports. Dans notre cas, l'on s'appesantira uniquement sur le domaine des problèmes environnementaux.
Divers problèmes que l'on rencontre au quotidien dans notre environnement peuvent être résolus et optimisés grâce à la programmation dynamique. C'est ainsi que dans l'article de Jingjing Zhao (enseignant d'informatique à l'université de Sciences et Technologie d'Hebei en Chine) intitulé <<The Application of dynamic programming in the system optimization of environmental problem>>, l'on retrouve divers problèmes environnementaux. Parmi ces derniers, deux seront principalement mis en avant pour notre étude :

## 1. LE PROBLEME DE L'ALLOCATION DES RESSOURCES D'EAU ENTRE DIVERS UTILISATEURS.

Ce problème sera écumé à travers un petit exemple.

Un réservoir d'eau est supposé avoir 7 unités et peut être alloué à 3 utilisateurs (A, B, C). Le bénéfice de chaque utilisateur avec différents volume d'eau est présenté dans le tableau 1. Déterminons le schéma optimal de distribution de l'eau aux trois utilisateurs qui maximise le bénéfice. L'on remarque que ce problème s'apparente à celui du sac à dos.

| Unités | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 0 | 5 | 15 | 40 | 80 | 90 | 95 | 100 |
| B | 0 | 5 | 15 | 40 | 60 | 70 | 73 | 75 |
| C | 4 | 4 | 40 | 40 | 50 | 50 | 51 | 53 |

Tableau 1 : Bénéfice en fonction des utilisateurs.

Ce problème peut être résolu en backtrackant la solution optimale généré graduellement par notre algorithme du sac à dos. L'application de cet algorithme nous indiquera qu'il serait judicieux d'offrir 4 unités d'eau à l'utilisateur A et 3 unités d'eau à l'utilisateur B pour obtenir un bénéfice de 120.

## 2. LE PROBLEME DU CHEMIN OPTIMUM POUR LE TRAITEMENT DES EAUX.

Le drainage quotidien des eaux usées d'une entreprise est d'environ 3000 T/j, la concentration en $DBO_5$ des eaux usées est de 1 500 mg / L (1,5 t / 1 000 t). Si l'effet du traitement sur trois types de mesures de transformation adoptées sont 65%, 85% et 90% respectivement, répondre aux questions suivantes :

(1) Comment traiter les eaux usées au moindre coût sur le principe de la réduction de la pollution ?

(2) Si les frais de pollution sont calculés par l'expression 10 * W / 0.1, donner un nouveau schéma optimal. (W représente la qualité de $DBO_5$ / T).

Ce problème s'apparente à celui de l'allocation des ressources avec la programmation dynamique. L'on ne le résoudra pas à cause des multiples traitements à effectuer avant de pouvoir appliquer notre algorithme mais quoiqu'il en soit, l'on observe tout de même l'importance de la programmation dynamique à travers cette application.

# CONCLUSION

En somme, pour notre thème de programmation dynamique nous avons abordé tout d'abord les grandes lignes de ce qu'est la programmation dynamique, après, nous avons étudié

quelques exemples, la comparaison de la programmation dynamique avec quelques autres méthodes algorithmiques en a suivi, et enfin, nous avons situé la programmation dynamique dans les domaines de recherche. Pour les différents exemples abordés, nous avons calculé les différentes complexités algorithmiques, celles-ci varient d'un problème à l'autre. Il en ressort que la programmation dynamique peut être vue comme une amélioration de la méthode diviser pour régner, typiquement conçu pour résoudre les problèmes d'optimisation. Elle permet d'obtenir à coup sûr une solution optimale contrairement à d'autre méthode algorithmique résolvant également le même type de problème en occurrence la méthode gloutonne. Pourquoi dit-on que la programmation dynamique est une méthode algorithmique bien trop lourde pour trouver les meilleures solutions d'un problème ?

## BIBLIOGRAPHIE


- **Introduction à l'algorithmique cours et exercices,** Charles Leiserson, Ronald Rivest, Clifford Stein, 1ère édition traduite de l'américain par Xavier Cazin
- **www.wikipedia.com**
- **www.openclassroom.com**

# PROGRAMMATION DYNAMIQUE

- The Application of dynamic programming in the system optimization of environmental problem, Jingjing Zhao
-

# PROGRAMMATION DYNAMIQUE