RÉPUBLIQUE DU CAMEROUN
REPUBLIC OF CAMEROON
Peace – Work - Fatherland

UNIVERSITÉ DE DSCHANG
UNIVERSITY OF DSCHANG
Scholae Thesaurus DschangensisIbiCordum
BP 96, Dschang (Cameroun) –
Tél. /Fax (237) 233 45 13 81
Website:http://www.univ-dschang.org.
E-mail : udsrectorat@univ-dschang.org

FACULTE DES SCIENCES
FACULTY OF SCIENCES

Département de Mathématiques et Informatique
Department of Mathematics and Computer Science
BP 67, Dschang (Cameroun)
Tél./Fax (237) …E-mail : faculte.sciences@univ-dschang.org

Informatique 4

Rapport de travail d'INF417 :
Complexité et algorithmes avancés

PROGRAMMATION DYNAMIQUE ET ALGORITHME GLOUTON

Travail réalisé par :

Noms	*Option*	*Matricule*
TABOUA MELONG SERAPHINE	**IA**	CM-UDS-18SCI0751
CECIL BONIVAN SAMAZO	**RSD**	CM-UDS-17SCI1808
TSIDIE TSOPGUE STEPHAN	**RSD**	CM-UDS-17SCI2629

Encadreur :
Dr KENGNE T. Vianney

Année académique :
2021-2022

PROGRAMMATION DYNAMIQUE ET ALGORITHMES GLOUTONS

SOMMAIRE

INTRODUCTION

Optimiser un problème, c'est déterminer les conditions dans lesquelles ce problème présente une caractéristique spécifique. Par exemple, déterminer le minimum ou le maximum d'une fonction est un problème d'optimisation. On peut également citer la répartition optimale de tâches suivant des critères précis, le problème du rendu de monnaie, le problème du sac à dos, la recherche d'un plus court chemin dans un graphe, le problème du voyageur de commerce, etc... Ainsi un problème d'optimisation est un problème de minimisation ou de maximisation d'une solution. De nombreuses techniques informatiques sont susceptibles d'apporter une solution exacte ou approchée à ces problèmes.

Certaines de ces techniques de résolution des problèmes d'optimisation comme l'énumération exhaustive de toutes les solutions (méthode naïve), ont un coût machine (complexité) qui les rend souvent peu pertinentes au regard de contraintes extérieures imposées (temps de réponse de la solution, ressources machines limités), et d'autres tel que les techniques programmations dynamique basées sur le principe de la DPR (diviser pour régner) et les algorithmes gloutons proposent des solutions raisonnables.

Tout au long de notre travail, nous nous attarderons tout d'abord sur le paradigme diviser pour régner (DPR), en suite nous présenterons le concept de la programmation dynamique puis le concept des algorithmes gloutons enfin clôturer par l'énumération de la comparaison entre la méthode de la programmation dynamique et d'autres méthode d'algorithmique.

I. LA METHODE DIVISER POUR REGNER(DPR)

a. DEFINITION

Le paradigme diviser pour régner est une méthode d'algorithmique basée sur le principe suivant : on prend un problème et on le divise en sous-problèmes. On résout les sous-problèmes (possiblement en les divisant de nouveau) et on combine les résultats pour obtenir la solution au problème original. C'est une approche de haut en bas

b. LE PRINCIPE

Le paradigme de programmation diviser pour régner nécessite que le problème à résoudre soit décomposable en sous-problème, cette décomposition pouvant être récursive, de nombreux algorithme ont une structure récursive : pour résoudre un problème donné, ils s'appellent récursivement une ou plusieurs fois sur des problèmes très similaires, mais de tailles moindres, résolvent les sous-problèmes de manière récursive puis combinent les résultats pour trouver une solution au problème initial.

Elle améliore considérablement le temps d'exécution par "division" en sous-problèmes indépendants dont les solutions sont recombinées.

3 conditions pour avoir un algorithme diviser-pour-régner efficace :

- Bien décider quand utiliser l'algorithme simple sur de petites ! Instances plutôt que les appels récursifs
- La décomposition d'une instance en sous-instances et la recombinaison des sous-solutions doivent être efficaces
- Les sous-instances doivent être autant que possible de même taille.

On s'attend ainsi à un **facteur logarithmique.**

Le paradigme **<< Diviser Pour Régner >>** donne lieu a trois étapes à chaque niveau de récursivité :

- **Diviser :** on décompose le problème initial en plusieurs sous-problèmes.
- **Régner :** on résoud chaque sous-problèmes récursivement.
- **Combiner :** on combine les solutions des sous-problèmes en une solution globale.

c. Algorithmes diviser-pour-régner – complexité

Souvent la complexité en temps d'un algorithme diviser-pour-régner sur une instance de taille *n* peut s'écrire comme:

$$T(n) = bT(n/b) + g(n)$$

Où g(n) est le temps prit pour casser et reconstruire.

Si on peut montrer que g(n) ϵ (n^k) pour un certain k alors le théorème suivant nous donne automatiquement la complexité de l'algorithme :

Soit $T: \mathbb{N} \longrightarrow \mathbb{R}^+$ une fonction éventuellement non décroissante telle que $T(n) = lT(\frac{n}{b}) + cn^k, \forall n > n_0$, où $\frac{n}{n_0}$ est une puissance de b. Alors,

$$T(n) \in \begin{cases} \theta(n^k) & si\ l < b^k \\ \theta(n^k log_b n) & si\ l = b^k \\ \theta(n^{log_b l}) & si\ l > b^k \end{cases}$$

d. EXEMPLE D'APPLICATION DE LA DPR: TRI PAR FUSION

1. Le principe

L'algorithme de tri par fusion est construit suivant le paradigme <<DPR>> :

- Il divise la séquence de n nombres à trier en deux sous-séquences de taille n/2.
- Il trie récursivement les deux sous-séquences.
- Il fusionne les deux sous-séquences triées pour produire la séquence complète triée.

La récursion termine quand la sous-séquence à trier est de longueur 1 … car une telle séquence est toujours triée.

2. Algorithme

La principale action de l'algorithme de tri par fusion est justement la fusion de deux listes triées.

❖ **La fusion**

Le principe de cette fusion est simple : à chaque étape, on compare les éléments minimaux des deux sous-listes triées, le plus petit des deux étant l'élément minimal de l'ensemble on le met de cote et on recommence. On conçoit ainsi un algorithme FUSIONNER qui prend en entrée un tableau A et trois entiers p, q et r, tels que p ≤ q < r et tels que les tableaux A[p..q] et A[q+1 ..r] soient triées. L'algorithme est présenté ci-dessous :

```
FUSIONNER (A, p, q, r)
i ← p                          indice servant à parcourir le tableau A [p.. q].
j ← q+1                        indice servant à parcourir le tableau A[q+1..r].
Soit C un tableau de taille r-p+1   tableau temporaire dans lequel on construit le résultat
k ← 1                          indice servant à parcourir le tableau temporaire.
tant que i ≤ q et j≤ r faire   boucle de fusion.
   Si A[i] < A[j] alors C[k] ←A[i]
                        I ← i+1
                  Sinon C[k] ←A[j]
                        j ← j+1
    k← k +1

tant que i ≤ q faire C[k] ← A[i]    on incorpore dans C les éléments de A [p..q] qui n'y
          i ← i+1                   seraient pas encore ; s'il y'en a, les éléments de
          k ← k+1                   A[q+1..r] sont déjà tous dans C.
tant que j ≤r faire C[k] ← A[j]     on incorpore dans C les éléments de A[q+1..r] qui
                    j ← j+1         n'y seraient pas encore ; s'il y'en a, les éléments
                    k ← k+1         de A[p..q] sont déjà tous dans C.
Pour k ←1 a r – p+1 faire           on recopie le résultat dans le tableau originel.
      A [p+ k-1] ←C[k]
```

Figure 1- Algorithme de fusion de deux sous-tableaux adjacents triés.

❖ **Complexité de la fusion**

Etudions les différentes étapes de l'algorithme :

- Les initialisations ont un cout constant de $\theta(1)$;
- Les boucles tant que de fusion s'exécutent au plus r - p fois, chacune de ses itérations étant de cout constant, d'où un cout total de O (r - p) ;
- Les deux boucles tant que complétant C ont une complexité respective au pire de q – p + 1 et de r – q, ces deux complexités étant de O (r - p) ;
- La recopie finale coute θ(r – p + 1).

Par conséquent, l'algorithme de fusion a une complexité en θ(r-p).

❖ **Le tri**

Ecrire l'algorithme de tri est maintenant une trivialité :

```
TRI-FUSION (A, p, r)
Si p < r alors q ← [(p + r) /2]
TRI-FUSION (A, p, q)
TRI-FUSION (A, q+1, r)
FUSIONNER (A, p, q, r)
```

Figure 2- Algorithme de tri par fusion.

3. Complexité

Pour déterminer la formule de récurrence qui nous donnera la complexité de l'algorithme TRI-FUSION, nous étudions les trois phases de cet algorithme ≪ **diviser pour régner**≫:
Diviser : cette étape se réduit au calcul du milieu de l'intervalle [p ; r]**,** sa complexité est donc en θ(1).
Régner : l'algorithme résout récursivement deux sous-problèmes de tailles respectives $\frac{n}{2}$**,** d'où une complexité en 2T($\frac{n}{2}$).
Combiner : la complexité de cette étape est celle de l'algorithme de fusion qui est en θ(n) pour la construction d'un tableau solution de taille n.
Par conséquent, la complexité du tri par fusion est donnée par la récurrence :

$$\mathbf{T(n)} = \begin{cases} \boldsymbol{\theta(1)} & si\ n = 1 \\ \boldsymbol{2T\left(\frac{n}{2}\right) + \theta(n)} & sinon \end{cases}$$

Pour déterminer la complexité du tri par fusion, nous utilisons **le Théorème de résolutions des récurrences** ≪**diviser pour régner**≫ qui stipule que :
Soient a ≥1 et b > 1 deux constantes, soit f(n) une fonction et soit T(n) une fonction définie pour les entiers positifs par la récurrence :

$$\mathbf{T(n) = aT\left(\frac{n}{b}\right) + f(n),}$$

Où l'on interprète $\frac{n}{b}$ soit comme $\lfloor \frac{n}{b} \rfloor$,soit comme $\lceil \frac{n}{b} \rceil$
T(n) peut être bornée asymptotiquement comme suit :

- ❖ Si $\mathbf{f(n) = \theta(n^{(\log_b a)} - \varepsilon)}$ pour une certaine constante $\varepsilon > 0$, alors, **T(n) =** $\theta(n^{\log_b a})$
- ❖ Si $\mathbf{f(n) = \theta(n^{\log_a b})}$**,** alors $\mathbf{T(n) = \theta(n^{\log_b a} \log n)}$**.**

Ici, a = 2 et b = 2, donc $\log_b a$ = 1 et nous nous nous trouvons dans le deuxième cas du théorème : f(n) = $\theta(n^{\log_b a}) = \theta(n)$. Par conséquent :
T(n) = θ(nlogn).

Pour les valeurs de n suffisamment grandes, le tri par fusion avec son temps d'exécution en **θ(nlogn)** est nettement plus efficace que le tri par insertion dont le temps d'exécution est en **$\theta(n^2)$.**

Illustration des étapes de l'algorithme avec les données d'un tableau A

Soit un tableau A =

38	27	43	3	9	82	10

Trions ce tableau par ordre croissant suivant le principe du tri par fusion.

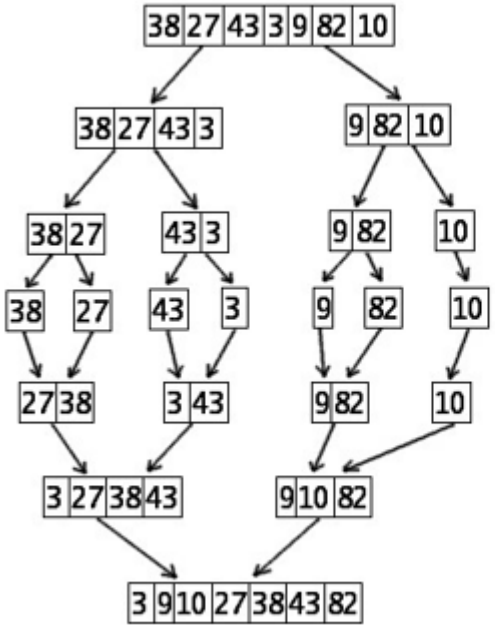


Figure 3 : illustraion des etapes du tri fusion avec un tableau

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

a- 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, les 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.

b. 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 de ses 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 mémorisation (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) s'épargnant ainsi le recalcule de la solution chaque fois que le sous-problèmes est rencontré.

Le développement d'un algorithme de programmation dynamique peut être typiquement planifié 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 partant des cas simples (cas d'arrêt des récursions) et en remontant progressivement jusqu'à l'énoncé du problème initial.

4. Construire une solution optimale à partir des informations calculées.

La dernière étape est utile si l'on souhaite calculer (avoir) 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.

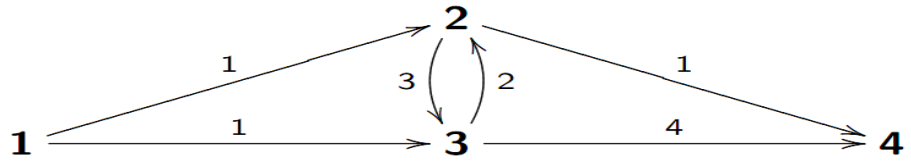
c. QUAND UTILISER LA PROGRAMMATION DYNAMIQUE ?

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-même 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 1 à 4 passe par 2, le tronçon de 1 à 2 de ce chemin n'est pas forcément le plus long chemin élémentaire de 1 à 2! 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.

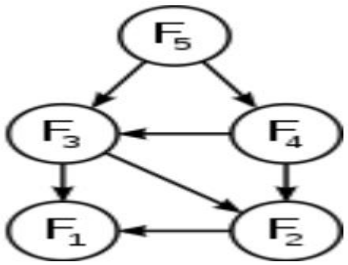


Figure 4 : Le graphe de dépendance des sous-problèmes pour calculer le terme de rang 5 de la suite de Fibonacci sv. Le calcul de F5 et F4 se chevauchent en F3, F2 et F1.

d. EXEMPLE D'APPLICATION : PROBLEME DU SAC A DOS

Notre premier 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
```

1- 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 ?

2- 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 ignorer 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 :

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

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].

e. PRODUITS MATRICIELS ENCHAINES

Notre deuxième 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 $\mathbf{M_1M_2...M_n}$.

Prérequis :

- Pour faire le produit $\mathbf{M_1M_2}$, 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 parenthèse** 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* $M_1M_2M_3M_4$ *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 $\mathbf{M_1M_2}...\mathbf{M_n}$ 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 $\mathbf{M_1M_2...M_n}$ (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 :

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

Résolvons cette équation de récurrence :

On a : P(n) = Ω (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.

i. 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 **i ≤ j**, la matrice résultante de l'évaluation du produit $\mathbf{M_iM_{i+1} \cdots M_j}$. Pour **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 **i ≤** *k* < **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 $\mathbf{M_{k+1}M_{k+2} \cdots M_j}$.

ii. 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 $\mathbf{M_i M_{i+1}} \cdot \cdot \cdot \mathbf{M_j}$ 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** $p_{i-1} \times p_k$ **x** 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 $\mathbf{M_i M_{i+1}} \cdot \cdot \cdot \mathbf{M_j}$ 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.

iii. *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 $C\frac{2}{n} + n = \Theta(n^2)$. 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 20×25.

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.

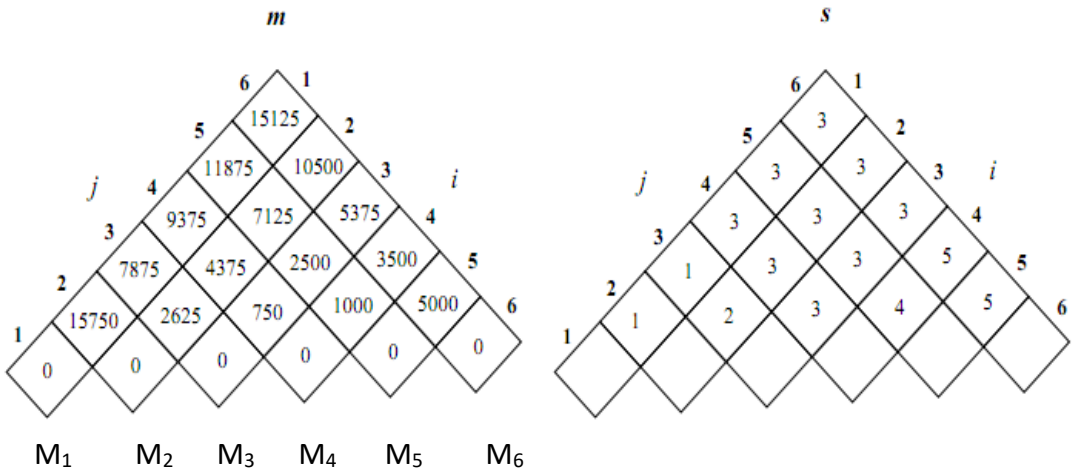


Figure 1 : Tableaux m et s calculés par ORDRE-CHAÎNE-MATRICES pour $n = 6$ et les dimensions des matrices M_1, M_2, M_3, M_4, M_5 et M_6 *sont* respectivement : $30 \times 35; 35 \times 15; 15 \times 5; 5 \times 10; 10 \times 20$ et 20×25.

iv. 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 mémoriser dans la 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 s , et ce à partir des matrices $\mathbf{M_i M_{i+1}} \cdots \mathbf{M_j}$, 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.**

f. LA METHODE GLOUTONNE

1. Définition et historique

Un algorithme glouton est un algorithme qui suit un principe permettant trouver un **optimum local** dans l'optique d'obtenir une solution **optimale globale** (du problème).

Un tel algorithme est celui qui suit **l'heuristique** de résolution de problème pour faire le choix localement optimal à chaque étape dans l'espoir trouver un optimum global. Dans beaucoup de problèmes, une stratégie gourmande ne produit pas toujours la solution optimale, mais une heuristique gourmande peut donner des solutions localement optimales qui en combinant se rapprochent d'une solution globalement optimale dans un délai raisonnable. Nous pouvons faire le choix qui nous semble mieux pour le moment, puis résoudre les problèmes qui se posent ultérieurement. Le choix fait par un **«** algorithme glouton peut dépendre des choix effectues jusqu'ici **mais pas des choix futurs** ». **I**l fait de manière itérative un choix gourmand après un autre, réduisant chaque problème donné en un problème plus petit.

2. Le principe de l'algorithme glouton et relativité d'optimisation

Les algorithmes gloutons sont des algorithmes pour lesquels, à chaque itération, on fixe la valeur d'une (ou plusieurs) des variables décrivant le problème sans remettre en cause les choix antérieurs.

Le principe est donc de partir d'une solution incomplète (éventuellement totalement indéterminée) que l'on complète de proche en proche en effectuant des choix localement optimaux. Dans certain cas, cela donnera finalement la meilleure solution : on parlera d'algorithmes gloutons exacts. Dans d'autres, on parlera d'heuristiques gloutonnes.

On appelle heuristique une méthode de résolution d'un problème qui ne repose pas sur l'examen détaillé du dit problème.

La **répétition** de cette stratégie très simple, permet de résoudre rapidement et de des problèmes d'optimisation sans avoir à tester toutes les possibilités. Cependant, l'algorithme glouton ne fournit pas toujours le meilleur résultat possible.
En ce qui concerne la relativité d'optimisation, nous notons que la solution obtenue par un algorithme glouton est le résultat d'une suite de choix gloutons, sans prises en compte des choix passés, ni anticipations des choix futurs. L'optimisation est donc potentiellement moindre qu'un algorithme effectuant une exploration systématique de toutes les possibilités.
Toutefois, les algorithmes gloutons sont généralement **moins couteux** qu'une exploration systématique. Ils sont ainsi capables de produire rapidement des résultats qui s'avèrent, dans la plupart des cas, suffisamment optimisés pour être acceptables.

3. Cas d'usages d'algorithmes gloutons

Les algorithmes goutons sont souvent employés pour résoudre les **problèmes d'optimisation.**
Un algorithme glouton construit une solution pas à pas sans revenir sur ses décisions, en effectuant à chaque étape le choix qui semble le meilleur, en espérant obtenir un résultat optimum global
Un algorithme glouton produit des solutions optimales si les deux propriétés suivantes sont vérifiées :
Propriété du choix glouton : on peut arriver à une solution globalement optimale en effectuant un choix localement optimal (glouton). Autrement dit : quand on considère le choix à faire, on fait le choix qui paraît le meilleur pour le problème courant, sans tenir compte des résultats des sous-problèmes.
Propriété de sous-structure optimale : un problème exhibe une ***sous-structure optimale*** si une solution optimale du problème contient les solutions optimales des sous-problèmes.
Soit S une solution optimale du problème P contenant le choix C, et $S' = S\{C\}$.
Pour résumer, on peut employer un algorithme glouton lorsque :

- Une *solution complète* peut être construite en passant par une succession de *solutions partielles*,
- Chaque *solution partielle* est établie en faisant un choix localement optimal.

Les approches gloutonnes se montrent efficaces pour :

- Déterminer le plus court chemin dans un graphe ;
- Compresser des données ;
- Organiser au mieux le parcours d'un voyageur visitant un ensemble de villes ;
- Organiser au mieux des plannings d'activités ou d'occupations des salles.

Les algorithmes gloutons sont simples dans leur logique et donc facile à implémenter, même si la détermination du choix glouton pertinent est plus ou moins évidente selon les cas.

4. Exemple d'application de l'algorithme glouton : rendu de la monnaie

Un achat dit en espèces se traduit par un échange de pièces et de billets. Les pièces désigneront dans la suite de l'expose les pièces ou les billets. Supposons qu'un achat induit

un rendu de **49 euros.** Quelles pièces peuvent être rendues ? La réponse, bien qu'étant évidente, n'est pas unique. Quatre pièces de **10 euros**, une pièce de **5 euros** et deux pièces de **2 euros** conviennent. Mais quarante-neuf pièces de 1 euros conviennent également ! Si la question est de rendre la monnaie avec un minimum de pièces possible, le problème change de nature. Mais la réponse reste simple : la première solution proposée. Toutefois, comment parvient-on à un tel résultat ? Quels choix ont été faits qui optimisent le nombre de pièces rendus ? C'est le problème de rendu de monnaie dont la solution dépend du système de monnaie utilisé.
Dans le système monétaire français, les pièces prennent les valeurs **1,2,5.10,20,50,100 euros**. Pour simplifier, nous nous intéressons uniquement sur les valeurs entières et oublions l'existence du billet de **500 euros**. Rendre **49 euros** avec un minimum de pièces est **un problème d'optimisation.** En pratique, sans s'en rendre compte généralement, tout individu met en œuvre un algorithme glouton. Il choisit d'abord la plus grande valeur de monnaie, parmi **1,2.5,10**, contenue dans **49 euros**. En occurrence quatre fois une pièce de **10 euros.** La somme de **9 euros** restant à rendre, il choisit une pièce de **5 euros**, deux pièces de **2 euros.** Cette stratégie gagnante pour la somme de **49 euros** l'est –elle pour n'importe quelle somme à rendre ? on peut montrer que la réponse est positive dans les systèmes monétaires français. Pour cette raison, un tel système de monnaie est qualifié de **canonique**. D'autres systèmes ne sont pas canoniques : l'algorithme glouton ne répond pas alors de **manière optimale.** Par exemple, avec le système {**1,3,6,12,24,30**}, l'algorithme glouton répond pas en proposant le rendu **49=30+12+6+1**, soit quatre pièces, alors que la solution optimale est **49= 2*24 +1**, soit trois pièces. La réponse à cette difficulté passe par **la programmation dynamique** qui est l'un des paradigmes de l'algorithme glouton.

4.1. Un algorithme glouton

Considérons un ensemble de **n** pièces de monnaie de valeurs :

$$v_1<v_2<\ldots<v_n$$

Avec $v_1=1$. On suppose que le système est **canonique**. On peut noter le système de pièces :

$$S_n= \{v_1 ; \ldots ; v_{n-1}\}.$$

Désignons par **s** la somme à rendre avec le minimum de pièces de S_n. L'algorithme glouton la plus grande valeur v_n et la compare à **s.**

- Si $s<v_n$, la pièce de valeur v_n peut pas être utilise. On reprend l'algorithme avec le système de pièces S_{n-1}.
- Si $s>=v_n$, la pièce v_n peut être utilisée pour une première fois. Ce qui fait une première pièce à comptabiliser, de valeur v_n, la somme restante à rendre étant $s-v_n$. L'algorithme continue avec le système de pièce S_n et cette nouvelle somme à rendre $s-v_n$.

L'algorithme est ainsi répété jusqu'à obtenir **une somme à rendre nulle**.
Remarque : il s'agit effectivement d'un algorithme glouton, la plus grande valeur de pièce étant systématiquement choisie si sa valeur est inférieure à la somme à rendre. Ce choix ne garantit rien de l'optimalité globale de la solution. Le choix fait est considéré comme pertinent et permet d'avancer plus avant dans le calcul. Toutefois, comme nous l'écrivons plus haut, si le système monétaire est canonique, alors la solution est optimale. Pour savoir si le système est canonique, **l'algorithme de kozen et zarks** apporte une réponse efficace.

4.2. Le code source

Définissons le système de pièce à l'aide d'un tableau des valeurs de pièces classées par valeurs croissante **S.**

```
# valeurs des pieces
systeme_monnaie= [1,2,5,10,20,50,100]
```

[numbers=none] pour stocker les pièces à rendre, une liste python initialement vide peut être utilisé.

```
# liste des pieces a rendre
lst_pieces=[]
```

La première pièce à rendre est potentiellement la dernière pièce du tableau système_monnaie. Une variable **i** de type entier est initialisée avec l'indice du dernier élément de ce tableau.

```
# l'indice de la premiere piece comparer a la somme a rendre
i= len(systeme_monnaie)- 1
```

Chaque fois qu'une pièce **S** n'est pas utilisable, la valeur de **i** sera diminuée de 1. Le programme s'arrête lorsque la valeur de **i** atteint 0. Ce qui mène à l'écriture d'une boucle conditionnelle pour remplir la liste des pièces choisies. La somme à rendre est initialement stocker dans la variable somme_a_rendre.

```
# somme a rendre
somme_a_rendre = 87
# boucle de construction de la liste des pieces
while somme_a_rendre > 0:
      valeur = systeme_monnaie[i]
      if somme_a_rendre < valeur :
        i -= 1
      else:
      lst_pieces.append(valeur)
      somme_a_rendre -= valeur
```

Pour finir, le code précèdent peut-être encapsuler dans une fonction qui reçoit deux arguments : la somme à rendre et le système de monnaie et qui renvoie la liste des pièces choisies par l'algorithme glouton.

```
def pieces_a_rendre(somme_a_rendre,systeme_monnaie):
# liste des pieces a rendre
lst_pieces=[]
# l'indice de la premiere piece comparer a la somme a rendre
i= len(systeme_monnaie)- 1
   while somme_a_rendre > 0:
       valeur = systeme_monnaie[i]
       if somme_a_rendre < valeur :
       i -= 1
      else:
      lst_pieces.append(valeur)
      somme_a_rendre -= valeur

      return lst_pieces
```

Il est annoté que, L'algorithme glouton est construit autour de trois principaux paradigmes parmi lesquels : la DPR (Diviser Pour Régner), la Programmation dynamique qui ont été détaillé plus haut ainsi les algorithmes gloutons (greedy algorithms en anglais).

g. 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. LA METHODE DE LA PROGRAMMATION DYNAMIQUE ET LA METHODE GLOUTONNE

Un algorithme glouton est un algorithme qui suit l'heuristique de résolution de problème pour faire de Le choix localement optimal à chaque étape avec l'espoir de trouver un optimum global. Il fait de manière itérative un choix gourmand après un autre, réduisant chaque problème donné en un problème plus petit. En d'autres termes, un algorithme glouton ne revient jamais sur ses choix.
C'est la principale différence avec la programmation dynamique, qui est exhaustive et qui garantit de trouver la solution. Après chaque étape, la programmation dynamique prend des décisions en fonction de toutes les décisions prises à l'étape précédente et peut reconsidérer le cheminement algorithmique de l'étape précédente vers la solution. Souvent, lorsque vous utilisez une méthode plus naïve, de nombreux sous-problèmes sont générés et résolus plusieurs fois. La méthode de programmation dynamique cherche à résoudre chaque sous-problème une seule fois, réduisant ainsi le nombre de calculs : une fois que la solution à un sous-problème donné a été calculée, elle est stockée ou "mémorisée » : la prochaine fois que la même solution est nécessaire, elle est simplement recherchée. Cette approche est particulièrement utile lorsque le nombre de sous-problèmes répétés augmente de manière exponentielle en fonction de la taille de l'entrée.

La programmation dynamique est fiable alors que les algorithmes gloutons ne sont pas toujours fiables
La principale différence entre la méthode Gourmand et la programmation dynamique est que **la décision (choix) prise par la méthode Gourmand dépend des décisions (choix)**

prises jusqu'à présent et ne repose pas sur des choix futurs ni sur toutes les solutions aux sous-problèmes. D'autre part, la programmation dynamique prend des décisions en fonction de toutes les décisions prises à l'étape précédente pour résoudre le problème.

Processus

De plus, une différence importante entre la méthode Gourmand et la programmation dynamique réside dans le fait que la méthode Gourmand fait d'abord un choix qui convient le mieux à l'époque, puis résout un sous-problème qui en résulte. La programmation dynamique résout tous les sous-problèmes, puis en sélectionne un qui aide à trouver la solution optimale.

La prise de décision

La méthode de prise de décision est une autre différence entre la méthode Gourmand et la programmation dynamique. La méthode Gourmand prend des décisions en considérant la première étape alors que la programmation dynamique prend des décisions à chaque étape.

Conclusion

La décision (choix) prise par la méthode Gourmand dépend des décisions (choix) prises jusqu'à présent et ne repose pas sur des choix futurs ni sur toutes les solutions aux sous-problèmes. Cependant, la programmation dynamique prend des décisions en fonction de toutes les décisions prises à l'étape précédente pour résoudre le problème. C'est la principale différence entre la méthode Gourmand et la programmation dynamique.

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 peut se résoudre graphiquement contrairement à la programmation dynamique.

CONCLUSION

En somme, pour notre thème de programmation dynamique et algorithme glouton, nous avons abordé tout d'abord les grandes lignes de ce qu'est du paradigme diviser pour régner, après, nous avons étudié un exemple d'illustration au nom du tri fusion, la méthode de la programmation dynamique ainsi qu'avec la présentation de la méthode gloutonne avec exemple a l'appui, et enfin, nous avons élaboré la comparaison entre la méthode de la programmation dynamique et d'autres méthodes d'algorithmique. 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**
- The Application of dynamic programming in the system optimization of environmental problem, Jingjing Zhao

Algorithmique et complexité avancés (Année 2021/2022)

TD – Programmation dynamique et algorithme glouton

Exercice 1 : **LA DISTANCE D'EDITION**

. . .aussi appelée *distance de Levenshtein*.

Soient x et y deux mots sur un alphabet, de longueurs respectives m et n.

On note $d(x, y)$ le nombre minimal d'insertions ou de suppressions de caractères pour passer de x à y. L'entier $d(x,y)$ est appelé *distance d'édition* entre les mots x et y.

Par exemple, on peut passer de mines à mimes des deux manières suivantes :

- mines → mies(suppression) → mimes(insertion)
- mines → mins(suppression) → mimns(insertion) → mimens(insertion)n→ mimes(suppression)

La première solution nécessite 2 insertions/suppressions, la seconde 4. Dans cet exemple, d (mines, mimes) = 2.

Questions.

1. Montrez que d établit bien une distance entre mots.
2. Montrez que $|m-n| \leq d(x,y) \leq m+n$.
3. Pour $i \in \{0,1,2,\ldots,m\}$ et $j \in \{0,1,2,\ldots,n\}$ on note x_i et y_j les préfixes de x et y de longueurs respectives i et j, avec la convention $x_0 = y_0 = \varepsilon$, où ε est le mot vide.

   Quelles sont les valeurs de $d(x_i, y_0)$ et $d(x_0, y_j)$?
4. Soient $i \in \{1,\ldots,m\}$ et $j \in \{1,\ldots,n\}$. Montrez que :
   $d(x_i, y_j) = \min\ d(x_{i-1}, y_{j-1}) + 2\,\delta_{i,j},\ d(x_{i-1}, y_j) + 1,\ d(x_i, y_{j-1}) + 1$,(1.1) où $\delta_{i,j}$ vaut 0 si la i-ème lettre de x et la j-ème lettre de y sont identiques, et 1 sinon.
5. Estimez la complexité de l'algorithme récursif implémentant « directement » l'équation (1.1). On pourra raisonner sur deux mots de même longueur n. Remarquez que dans cet algorithme les mêmes calculs sont faits plusieurs fois.
6. Déduisez de l'équation (1.1) un algorithme de calcul de $d(x, y)$ tel que le nombre d'opérations et l'occupation mémoire soient en $O(mn)$.
7. Calculez d(ingénieur, ingéneur).
8. Trouvez une suite d'opérations réalisant le nombre minimal d'opérations dans le cas précédent.

Remarque : On définit aussi la distance d'édition comme le nombre minimal d'insertions, suppressions ou substitutions (ces dernières sont bien sûr possibles dans le cadre de l'énoncé, mais « coûtent » ici 2 opérations). On peut aussi donner des coûts

différents à chaque opération. Le principe de la résolution par programmation dynamique reste le même.

La distance d'édition est utilisée dans de nombreux contextes (par exemple les correcteurs d'orthographe des logiciels de traitement de texte). Une variante est également utilisée dans les problèmes d'alignement de génomes en biologie moléculaire.

Exercice 2

On cherche à sélectionner **cinq** nombres de la liste suivante en cherchant à avoir leur somme la plus grande possible (maximiser une grandeur) et en s'interdisant de choisir deux nombres voisins (contrainte).

15 - 4 - 20 - 17 - 11 - 8 - 11 - 16 - 7 - 14 - 2 - 7 - 5 - 17 - 19 - 18 - 4 - 5 - 13 - 8

1. quelle est la stratégie gloutonne à mettre en œuvre ?
2. Appliquez cet algorithme glouton sur le tableau.
3. Vérifiez que {20,18,17,16,14} est une autre solution possible.
4. Que dire de la solution gloutonne ?
5. Evaluer sa complexité.

Exercice 3: Problème de location de skis

On cherche à attribuer m paires de skis (de longueurs $l_1, l_2, \ldots, l_m$) à n skieurs (de tailles $t_1, t_2, \ldots, t_n$), de manière à minimiser la somme des différences entre la longueur des skis et la taille des skieurs.

Bien sûr, $n \leq m$.

On suppose sans perte de généralité que les l_i et les t_i sont classés par ordre croissant.

Le problème consiste à trouver une fonction (d'allocation) a telle que le skieur n^o i se voit attribuer la paire de skis n^o $a(i)$ et qui minimise :

$$\sum_{i=1}^{n} \left| t_i - l_{a(i)} \right|$$

La même paire de skis n'est pas attribuée à deux skieurs, donc la fonction a est injective.

Questions.

1. Montrez que l'optimum est atteint par une fonction a croissante. On supposera donc que l'affectation des skis des i premiers skieurs se fait parmi les j premières paires de skis.
2. Soit $S(i, j)$ le minimum de la fonction-objectif du problème restreint aux i premiers skieurs et j premières paires de skis.
   Montrez que :

$$S(i,j) = \min\{S(i,j-1), S(i-1,j-1) + |t_i - l_j|\}.$$

3. Comment procéderiez-vous pour calculer la solution optimale ?
4. Quelle est la complexité de cet algorithme ?

Exercice 4: Problème du voyageur de commerce

Un voyageur a ciblé plusieurs villes qu'il souhaite visiter. Il cherche un itinéraire passant par toutes ces villes et qui minimise la distance totale parcourue (minimiser une grandeur). Les villes peuvent être visitées dans n'importe quel ordre mais aucune ne doit être négligée, et le visiteur doit revenir à la fin à sa ville de départ.
Le voyageur part de Nancy et souhaite visiter Metz, Paris, Reims et Troyes, avant de retourner à Nancy.
Voici un tableau donnant les distances kilométriques entre chacune de ces villes.

	Nancy	**Metz**	**Paris**	**Reims**	**Troyes**
Nancy		55	303	188	183
Metz	55		306	176	203
Paris	303	306		142	153
Reims	188	176	142		123
Troyes	183	203	155	123	

1. Quelle est la stratégie gloutonne à mettre en œuvre ?
2. Mettez-en œuvre cette stratégie et donnez la solution.
3. Calculez la distance totale pour le parcours Metz - Reims - Paris - Troyes (départ et arrivée à Nancy sous-entendus)
4. Que dire de la solution gloutonne ?

Exercice 5: Problème du sac à dos

On rappelle qu'il y a plusieurs stratégies gloutonnes pour donner une solution à ce problème
Stratégie 1 : prendre toujours l'objet de plus grande valeur n'excédant pas la capacité restante (il faut trier préalablement par valeur décroissante) Stratégie 2 : prendre toujours l'objet de plus faible masse (il faut trier préalablement par masse croissante).
Stratégie 3 : prendre toujours l'objet de plus grand rapport n'excédant pas la capacité restante (il faut trier préalablement par rapport décroissant)

Exemple 1
Considérons les objets suivants et un sac de capacité maximale 2 kg.

Objet	**A**	**B**	**C**
Masse (**en kg**)	1,5	2	0,3
Valeur (en **€**)	200	500	400
Valeur /masse	133,33…	250	1333,33…

1. Appliquez chacune des stratégies à ce problème.
2. Quelle est la meilleure stratégie dans ce cas ?
3. Listez toutes les combinaisons possibles, puis celles respectant la contrainte et déduisez-en la solution optimale au problème. Comparez-la aux stratégies gloutonnes.
4. Evaluer la complexité de cet algorithme.

Exemple 2
On considère désormais les objets suivants et un sac de capacité maximale 5 kg. Les rapports valeur/masse ont été calculé et arrondi à 10^{-1} prés.

Objet	Masse (en **kg**)	Valeur (en milliers d'**€**)	Rapports
A	4,57	114	24,9
B	0,63	32	50,8
C	1,65	20	12,1
D	0,085	4	47,1
E	2,15	18	8,4
F	2,71	80	29,5
G	0,32	5	15,6

1. Appliquez chacune des stratégies à ce problème.
2. Quelle est la meilleure stratégie dans ce cas ?
3. Le sac est-il une solution au problème ? Quelle est sa valeur ? Que dire des solutions gloutonnes ?
4. Evaluer la complexité de cet algorithme