

Chapitre 7 : Éléments d'algorithmique

Table des matières

1	Introduction	2
2	Diviser pour régner	2
2.1	Méthode et premiers exemples	2
2.1.1	Méthode	2
2.1.2	Remarque	2
2.1.3	Complexité des algorithmes <i>diviser pour régner</i>	2
2.1.4	Exemple (Tri fusion)	2
2.1.5	Exemple (exponentiation rapide)	3
2.1.6	Exemple (recherche dichotomique)	3
2.1.7	Exemple (Tri rapide)	4
2.2	Application : calcul de la médiane	4
2.2.1	Problème	4
2.2.2	Solution naïve	4
2.2.3	Généralisation et algorithme <i>diviser pour régner</i>	4
2.2.4	Algorithme de la médiane des médianes	5
2.3	Application : couverture de points par des segments de même longueur	7
2.3.1	Problème	7
2.3.2	Remarque	7
2.3.3	Résolution par dichotomie du problème d'optimisation	7
2.3.4	Résolution du problème de décision	7
2.3.5	Analyse de complexité	9
3	Programmation dynamique	9
3.1	Méthode	9
3.1.1	Introduction	9
3.1.2	Approche descendante de la programmation dynamique	9
3.1.3	Approche ascendante	10
3.1.4	Exemple	11
3.2	Application aux problèmes d'optimisation	11
3.2.1	Introduction	11
3.2.2	Exemple	11
3.2.3	Exemple : Produit de matrices	12

1 Introduction

À la manière de la recherche d'un élément dans un ABR, on s'intéresse à la décomposition d'un problème en un ou plusieurs sous-problèmes. On distingue plusieurs méthodes selon le type de problème à résoudre, la manière dont on décompose le problème et la manière dont on reconstruit la solution au problème à partir de celles des sous-problèmes. Nous étudierons la méthode *diviser pour régner*, la programmation dynamique, et les algorithmiques gloutons.

2 Diviser pour régner

2.1 Méthode et premiers exemples

2.1.1 Méthode

La méthode *diviser pour régner* se décompose en 3 étapes :

- (1) Diviser le problème en un ou plusieurs sous-problèmes **disjoints** ;
- (2) Résoudre les sous-problèmes, soit récursivement, soit directement si le sous-problème est assez simple ;
- (3) Reconstruire la solution au problème initial à partir de celles des sous-problèmes.

2.1.2 Remarque

On retrouve l'idée des principes d'induction, où l'on utilise l'hypothèse d'induction (la solution à un sous-problème) pour démontrer la propriété sur une instance plus grande (le problème initial).

2.1.3 Complexité des algorithmes *diviser pour régner*

Cette méthode donne une relation de récurrence de la forme

$$C(n) = C_{\text{décomposition}}(n) + \sum_{p \text{ sous-problèmes}} C(|p|) + C_{\text{reconstruction}}(n)$$

(ici, $|p|$ désigne la taille du problème p)

L'enjeu est de trouver des manières peu coûteuses de décomposer / reconstruire et d'obtenir peu de sous-problèmes assez petits pour avoir une complexité meilleure que celle de la résolution naïve du problème.

2.1.4 Exemple (Tri fusion)

Tri fusion (*cf* chap 1, 3.1.4)

Problème : on veut trier une liste



Algo :

- (1) Diviser : on décompose la liste en 2 listes de tailles égales à une unité près ;
- (2) Résoudre : on trie récursivement les 2 listes, sauf si elles sont de taille ≤ 1 (déjà triées)
- (3) Reconstruire : on fusionne les 2 listes triées en une liste.

Récurrance :

$$C(n) = \mathcal{O}(n) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n)$$

Donc $\mathcal{O}(n \log n)$

2.1.5 Exemple (exponentiation rapide)

(cf TD₀₄)

Problème : calcul de x^n

Algo :

- (1) Diviser : on veut calculer $x^{\lfloor \frac{n}{2} \rfloor}$;
- (2) Résoudre : appel récursif, ou si $\lfloor \frac{n}{2} \rfloor = 0$, on renvoie 1 ;
- (3) Reconstruire : si $n \equiv 0 \pmod{2}$, on calcule $x^{\lfloor \frac{n}{2} \rfloor} \cdot x^{\lfloor \frac{n}{2} \rfloor}$, sinon on calcule $x \cdot x^{\lfloor \frac{n}{2} \rfloor} \cdot x^{\lfloor \frac{n}{2} \rfloor}$.

Récurrance :

$$C(n) = \mathcal{O}(1) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \mathcal{O}(1)$$

Donc $\mathcal{O}(\log n)$.

2.1.6 Exemple (recherche dichotomique)

(cf TD₀₆)

Problème : on veut déterminer si une valeur apparaît dans un tableau trié.

Algo :

- (1) Diviser : on s'intéresse à la moitié gauche ou droite du tableau selon la comparaison de la valeur recherchée avec l'élément au milieu du tableau ;
- (2) Résoudre : appel récursif sauf si le sous-tableau est de taille ≤ 1 ou si l'élément au milieu est la valeur recherchée ;
- (3) Reconstruire : la solution au sous-problème est la solution au problème initial.

Récurrance :

$$C(n) = \mathcal{O}(1) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

Donc $\mathcal{O}(\log n)$.

2.1.7 Exemple (Tri rapide)

(cf colle 01)

Problème : trier une liste.

Algo :

- (1) Diviser : on choisit un élément “pivot” et on répartit la liste en 2 sous-listes (celle des éléments $<$ au pivot et celle des éléments $>$ au pivot) et on compte le nombre d’occurrences du pivot ;
- (2) Résoudre : appel récursif sauf si la liste est de taille ≤ 1 ;
- (3) Recomposition : on concatène les 2 listes triées (dans le bon ordre) en plaçant entre les 2 le bon nombre d’occurrences du pivot.

Réurrence :

$$C(n) = \mathcal{O}(1) + C(q) + C(r) + \mathcal{O}(p + q)$$

où $\left\{ \begin{array}{l} p \text{ est le nombre d'occurrences du pivot} \\ q \text{ est le nombre d'éléments } < \text{ pivot} \\ r \text{ est le nombre d'éléments } > \text{ pivot} \end{array} \right.$

Avec un choix déterministe en temps constant du pivot (ex : le premier élément), on a :

$\mathcal{O}(n^2)$ dans le pire cas ;

$\mathcal{O}(n \log n)$ dans le meilleur cas.

2.2 Application : calcul de la médiane

2.2.1 Problème

On veut calculer la médiane d’une liste de n valeurs distinctes, c’est-à-dire l’élément de rang $\left\lfloor \frac{n}{2} \right\rfloor$ où le rang d’un élément est le nombre d’éléments qui lui sont strictement inférieurs.

2.2.2 Solution naïve

On trie la liste et on renvoie l’élément au milieu

Complexité : $\mathcal{O}(n \log n)$ (ex : avec le tri fusion)

Objectif : $\mathcal{O}(n)$ (optimal puisqu’on doit bien lire tous les éléments)

2.2.3 Généralisation et algorithme *diviser pour régner*

On s’intéresse au problème suivant : trouver l’élément de rang i , où i est un paramètre.

Algo :

- (1) Diviser : on choisit un pivot p et on détermine les listes $l_<$ et $l_>$ des éléments strictement inférieurs / supérieurs à p ;



(2) Résoudre : cela dépend de la taille de $l_{<}$:

- Si $|l_{<}| > i$, alors on cherche l'élément de rang i dans $l_{<}$;
- Si $|l_{<}| = i$, alors p est l'élément de rang i ;
- Si $|l_{<}| < i$, alors on cherche l'élément de rang $i - |l_{<}| - 1$ dans $l_{>}$;

(3) Reconstruire : l'élément de rang i est le résultat de la résolution du sous-problème.

Récurrence :

$$C(n) = \mathcal{O}(n) + C_{\text{choix pivot}}(n) + \begin{cases} C(q) \\ \text{ou} \\ \mathcal{O}(1) \\ \text{ou} \\ C(r) \end{cases} \quad \text{selon la taille de } l_{<}$$

avec les notations de 2.1.7.

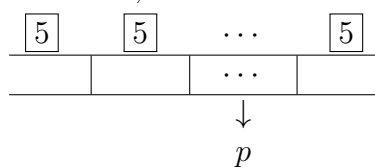
Dans le pire cas, comme pour le tri rapide $\mathcal{O}(n^2)$ par exemple si on choisit le premier élément comme pivot et si le tableau est trié et $i = n - 1$.

Dans le meilleur cas, le choix du pivot est déterministe. On peut se rapprocher du meilleur cas de tri rapide en choisissant un pivot proche de la médiane.

On peut s'approcher de la médiane (que l'on cherche à calculer), à moindre coût à l'aide de l'algorithme de la médiane des médianes.

2.2.4 Algorithme de la médiane des médianes

Idée : on regroupe les éléments de la liste en petits paquets (de taille constante) dont on peut calculer les médianes naïvement, et on choisit comme pivot la médiane de ces médianes, calculée récursivement.



Complexité :

$$C(n) = \underbrace{\mathcal{O}(n) + C\left(\frac{n}{5}\right)}_{\text{calcul du pivot}} + \underbrace{\mathcal{O}(n)}_{\text{division}} + \underbrace{f(n)}_{\text{éventuel appel récursif}}$$

On peut majorer la valeur de $f(n)$ en majorant les tailles des listes $l_{<}$ et $l_{>}$.

On s'intéresse à $l_{<}$ (le cas de $l_{>}$ se traite de manière similaire).

Observons les $\frac{n}{5}$ médianes des paquets. Parmi celles-ci, il y en a la moitié qui sont inférieures à p . Dans le pire cas, tous les éléments des $\frac{n}{10}$ paquets de ces médianes sont inférieures à p . Dans les autres paquets, il y a au plus 2 éléments inférieure à p (car la médiane et les 2 éléments qui lui sont supérieurs sont supérieurs à p).

Au total, il y a au plus $5 \cdot \frac{n}{10} + 2 \cdot \frac{n}{10} = \frac{n}{2} + \frac{n}{5} = 7 \frac{n}{10}$ éléments dans $l_{<}$.

Donc $C(n) \leq C\left(\frac{n}{5}\right) + C\left(\frac{7n}{10}\right) + \mathcal{O}(n)$

En pratique, sous les approximations, on montre

$$\exists c > 0 \mid \forall n, C(n) \leq cn + C\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + C\left(7 \left\lfloor \frac{n}{10} \right\rfloor + 4\right)$$

Montrons que $C(n) = \mathcal{O}(n)$, i.e

$$\exists c' > 0, \exists N \in \mathbb{N} \mid \forall n \geq N, C(n) \leq c'n$$

On procède par analyse-synthèse :

– Analyse :

$$\begin{aligned} C(n) &\leq cn + C\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + C\left(7 \left\lfloor \frac{n}{10} \right\rfloor + 4\right) \\ &\leq cn + c' \left\lfloor \frac{n}{5} \right\rfloor + c' \left(7 \left\lfloor \frac{n}{10} \right\rfloor + 4\right) \\ &\leq cn + \frac{c'n}{5} + c' \left(\frac{7n}{10} + 4\right) \\ &\leq cn + c' \left(\frac{9n}{10} + 4\right) \\ &\leq c'n \qquad \qquad \qquad \text{dès que } cn \leq c' \left(\frac{n}{10} - 4\right) \end{aligned}$$

$$\text{Or } cn \leq c' \left(\frac{n}{10} - 4\right) \Leftrightarrow \frac{10cn}{n-40} \leq c' \text{ (si } n > 40)$$

$$\text{et } \frac{n}{n-40} \leq 2 \text{ si } n \geq 80$$

– Synthèse :

On choisit $N = 800$ et $c' = 20c$

$$\forall n \geq N, \frac{n}{n-40} \leq 2 \text{ donc } \frac{10cn}{n-40} \leq 20c = c'$$

Donc une récurrence avec la même démonstration qu'en analyse conclut :

$$C(n) \leq c'n$$

Donc l'algorithme de la médiane des médianes permet de déterminer la médiane en temps linéaire.



2.3 Application : couverture de points par des segments de même longueur

2.3.1 Problème

Étant donné n points x_1, \dots, x_n sur la droite réelle, et $k \in \mathbb{N}^*$, on veut déterminer la longueur l minimale telle qu'il existe une couverture de x_1, \dots, x_n par k segments de longueur l .

k segments $[l_1, r_1], \dots, [l_k, r_k]$ forment une couverture de x_1, \dots, x_n si, et seulement si :

$$\forall i \in \llbracket 1 ; n \rrbracket, \exists j \in \llbracket 1 ; k \rrbracket \mid x_i \in [l_j, r_j]$$

2.3.2 Remarque

Ceci est un problème d'optimisation : on cherche à minimiser / maximiser une certaine quantité en respectant des contraintes.

On peut en général associer à un problème d'optimisation un problème de décision, c'est-à-dire un problème où l'on se pose la question de l'existence d'un objet vérifiant des contraintes.

Ici : étant donné x_1, \dots, x_n , k et l , existe-il une couverture de x_1, \dots, x_n par k segments de longueur l ?

2.3.3 Résolution par dichotomie du problème d'optimisation

Idée : on suppose que l'on sait résoudre le problème de décision et on procède à une recherche dichotomique du l minimal pour lequel l'algorithme de résolution du problème de décision répond *oui*.

On sait que $L = \frac{\max_{i,j} |x_i - x_j|}{k}$ permet de construire une couverture, donc on cherche dans $[0, L]$.

On procède à une recherche dichotomique dans un tableau virtuel (on ne le construit pas) qui à chaque $i \in \llbracket 0 ; L \rrbracket$ associe la réponse au problème de décision pour $l = i$.

2.3.4 Résolution du problème de décision

Idée : on s'intéresse au problème de décision dual suivant : étant donné n segments $[l_1, r_1], \dots, [l_n, r_n]$ et $k \in \mathbb{N}^*$, peut-on choisir k points x_1, \dots, x_k tels que

$$\forall i \in \llbracket 1 ; n \rrbracket, \exists j \in \llbracket 1 ; k \rrbracket \mid x_j \in [l_i, r_i]$$

Cela permet de résoudre le problème initial : si $\exists ([l_i, r_i])_{i \in \llbracket 1 ; n \rrbracket}$ couverture de $(x_j)_{j \in \llbracket 1 ; n \rrbracket}$ par des segments de longueur l , alors on a une solution au problème dual pour les intervalles $[x_1 - l, x_1], \dots, [x_n - l, x_n]$ ($x_i \in [l_i, r_i] \Leftrightarrow l_j \in [x_i - l, x_i]$)

Résolution du problème dual : soit $([l_i, r_i])_{i \in \llbracket 1 ; n \rrbracket}$ n segments et $k \in \mathbb{N}^*$.

$$[\left[\begin{array}{c}] \\] \end{array} \right]]]$$

On traite les segments par extrémités droite croissante :

Algorithme :

$P \leftarrow \emptyset$

Trier et renuméroter les segments par r_i croissant

$p \leftarrow r_1$

$P \leftarrow P \cup \{p\}$

Pour i de 2 à n :

Si $l_i > p$

$p \leftarrow r_i$

$P \leftarrow P \cup \{p\}$

Si $|P| \leq k$ succès, sinon échec.

Correction : tous les arguments sont rencontrés par un point de P .

Invariant de boucle : au début du tour d'indice i , tous les segments jusqu'à l'indice $i - 1$ sont rencontrés et $p = \max P = r_j$ pour un certain $j < i$

• Initialement : $P = \{r_1\} = \{p\}$ donc $p = \max P = r_1$ avec $1 < 2$, et $r_1 \in [l_1, r_1]$ donc $[l_1, r_1]$ est rencontré par un élément de P

• Conservation de l'invariant : on suppose l'invariant vrai au début du tour i , en particulier $p = \max P = r_j$ pour $j < i$

Si $l_i \leq p$, comme $j < i$, $r_j \leq r_i$, donc $l_i \leq p \leq r_i$ et il est inutile de rajouter un point pour $[l_i, r_i]$. De plus, $p = \max P = r_j$ avec $j < i + 1$

Si $l_i > p$, il faut un point supplémentaire.

$P \cup \{r_i\}$ rencontre bien $[l_i, r_i]$ et $p = r_i = \max P \cup \{r_i\}$ avec $i < i + 1$

Optimalité : en notant $P = \{p_i \mid i \in \llbracket 1 ; |f| \rrbracket\}$, on montre par récurrence finie que $\forall i \in \llbracket 1 ; |P| \rrbracket, \exists P_{\text{opt}}$ optimal rencontrant tous les segments et contenant $\{p_j \mid j \in \llbracket 1 ; i \rrbracket\}$

Initialisation : $i = 1$

On considère P_{opt} une solution optimale.

Si $p_1 \notin P_{\text{opt}}$, on s'intéresse à $\min P_{\text{opt}}$, qui est nécessaire pour rencontrer les segments $[l_1, r_1], \dots, [l_j, r_j]$ pour un certain j .

$\forall k \leq j$, on remarque que $l_k \leq \min P_{\text{opt}} \leq r_1 = p_1 \leq r_k$ donc $P_{\text{opt}} \setminus \{\min P_{\text{opt}}\} \cup \{r_1\}$ convient.

Hérédité : par hypothèse de récurrence, $\exists P_{\text{opt}}$ qui rencontre $[l_1, r_1], \dots, [l_n, r_n]$ et qui contient $\{p_s \mid s \in \llbracket 1 ; i \rrbracket\}$.

On numérote les éléments de P_{opt} : $p_{\text{opt}1}, \dots, p_{\text{opt}|P_{\text{opt}}|}$, si $p_{\text{opt}i+1} \neq p_{i+1}$, on utilise le même raisonnement que ci-dessus pour montrer que tous les intervalles pour lesquels $p_{\text{opt}i+1}$ est nécessaire sont rencontrés par p_{i+1} .

Pour la même raison, on peut remplacer $p_{\text{opt}i+1}$ par p_{i+1} .



Remarque : on utilise un algorithme glouton pour résoudre le problème dual (cf ??)

2.3.5 Analyse de complexité

- Résolution du problème dual : $\mathcal{O}(n \log n)$ à cause du tri ;
- Conversion du problème de décision en son dual : $\mathcal{O}(n)$ pour la construction des $[x_i - l, x_i]$;
- Recherche par dichotomie : calcul du diamètre $D = \max_{i,j} |x_i - x_j|$ en $\mathcal{O}(n)$, puis $\mathcal{O}\left(\log \frac{D}{k}\right)$ itérations ;
- Au total : $\mathcal{O}\left(n \log(n) \log \frac{D}{k}\right)$

3 Programmation dynamique

3.1 Méthode

3.1.1 Introduction

Si les sous-problèmes ne sont pas disjoints, la méthode *diviser pour régner* donne des algorithmes dont la complexité est mauvaise, parce que l'on est amené à résoudre plusieurs fois le même sous-problème.

Exemple : calcul des coefficients binomiaux avec le triangle de Pascal :

$$\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$$

L'implémentation naïve de cette formule donne une complexité

$$\begin{aligned} C(n, p) &= C(n-1, p) + C(n-1, p-1) \\ &= \dots \\ &= \mathcal{O}\left(\binom{n}{p}\right) \end{aligned}$$

exponentiel dans le pire cas $\left(p = \frac{n}{2}\right)$

3.1.2 Approche descendante de la programmation dynamique

On stocke les valeurs calculées pour retrouver en temps constant si on en a encore besoin. On parle de *mémoïsation*.

Exemple :

```

1 | let binom_memoised (n : int) (p : int) : int =
2 |   let t = Array.make_matrix (n + 1) (p + 1) (-1) in
3 |   let rec aux (i : int) (j : int) : int =
4 |     if t.(i).(j) <> -1 then t.(i).(j)
5 |     else
6 |       if j = 0 || i = j then begin
7 |         t.(i).(j) <- 1;
8 |         1
9 |       end else begin
10 |         t.(i).(j) <- aux (i - 1) j + aux (i - 1) (j - 1);
11 |         t.(i).(j)
12 |       end
13 |   in aux n p;;

```

3.1.3 Approche ascendante

On adopte un point de vue impératif en inversant l'ordre dans lequel les sous-problèmes sont considérés. On commence par les cas de base et on construit progressivement la solution au problème global.

Remarque : on doit faire attention aux dépendances entre les valeurs calculées car elles ne sont plus gérées par la pile d'exécution.

Exemple : triangle de Pascal :

$$i \rightarrow \begin{pmatrix} j \\ \nwarrow \nearrow \end{pmatrix}$$

$$\binom{i}{j} = \binom{i-1}{j} + \binom{i-1}{j-1} \rightarrow \text{on remplit la matrice ligne par ligne}$$

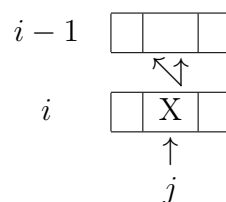
```

1 | let binom_ascending (n : int) (p : int) : int =
2 |   let t = Array.make_matrix (n + 1) (p + 1) 1 in
3 |   for i = 2 to n do
4 |     for j = 1 to min (n - 1) p do
5 |       t.(i).(j) <- t.(i - 1).(j) + t.(i - 1).(j - 1)
6 |     done
7 |   done
8 |   t.(n).(p)

```

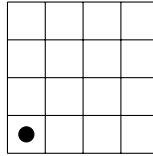
Complexité spatiale / temporelle : $\mathcal{O}(np)$

On peut faire mieux en complexité spatiale : $\mathcal{O}(p)$: un seul tableau de taille p suffit



3.1.4 Exemple

Nombres de Delannoy : nombre de chemins distincts que peut prendre une dame sur un échiquier pour aller du coin inférieur gauche au coin supérieur droit en utilisant que les déplacements par rangée / colonne croissante, en fonction des dimensions de l'échiquier.



$$D(i, j) = \underbrace{D(i, j-1)}_{\rightarrow} + \underbrace{D(i-1, j)}_{\uparrow} + \underbrace{D(i-1, j-1)}_{\nearrow}$$

Si $i \neq 0$ et $j \neq 0$

$D(i, j) = 1$ si $i = 0$ ou $j = 0$

par programmation dynamique, on calcule $D(n, p)$ en $\mathcal{O}(np)$ en temps et $\mathcal{O}(p)$ en espace (2 rangées suffisent).

3.2 Application aux problèmes d'optimisation

3.2.1 Introduction

Un problème d'optimisation est un problème dans lequel on veut maximiser ou minimiser une valeur sous certaines contraintes. L'objectif est de déterminer les paramètres permettant d'atteindre l'optimum. La méthode de la programmation dynamique s'applique à de nombreux problèmes d'optimisation. Pour appliquer cette méthode, le problème doit avoir une certaine propriété, appelée *propriété de sous-structure optimale* : si l'on dispose d'une solution optimale au problème, alors cette solution induit une solution optimale à un ou plusieurs sous-problèmes.

Par exemple, si un plus court chemin de A vers B passe par C , alors le sous-chemin de A vers C est bien un plus court chemin de A à C .

3.2.2 Exemple

Chemin de poids maximum sur un échiquier.

On reprend le problème vu en 3.1.4, mais on ne cherche pas à déterminer le nombre de chemins, mais plutôt on suppose que les cases de l'échiquier contiennent des poids et on veut trouver un chemin qui maximise la somme des poids des cases parcourues.

On modélise le problème à l'aide d'une matrice de poids :

$$\begin{pmatrix} 31 & 21 & 12 & 26 & 34 \\ 37 & 21 & 34 & 26 & 10 \\ 2 & 39 & 12 & 49 & 47 \end{pmatrix}$$

Une recherche exhaustive n'est pas raisonnable : le nombre de chemins à tester pour une matrice $n \times p$ est le nombre de Delannoy $D(n, p)$. Or $D(n, p) \geq \binom{n+p}{p}$ (en interdisant les pas diagonaux, il faut n pas verticaux, et p pas horizontaux pour passer d'un coin à l'autre. Un chemin est donc caractérisé par l'emplacement de ses pas horizontaux).

Vérifions que ce problème satisfait la propriété de sous-problème optimale : si un chemin de poids maximal de $(0, 0)$ à $(n-1, p-1)$ passe par (i, j) , alors le sous-chemin de $(0, 0)$ à (i, j) est de poids maximal. Sinon, on complète un chemin de poids maximal de $(0, 0)$ à (i, j) par le sous-chemin de (i, j) à $(n-1, p-1)$ et on obtient un chemin valide de $(0, 0)$ à $(n-1, p-1)$ qui contredit l'optimalité du chemin initial.

La propriété de sous-structure optimale correspond en fait à une récurrence sur le poids maximal $w_{i,j}$ d'un chemin de $(0, 0)$ à (i, j) : on note $M = (m_{i,j})_{i,j}$ la matrice définissant le problème.

Alors :

$$w_{i,j} = m_{i,j} + \begin{cases} 0 & \text{si } i = j = 0 \\ w_{0,j-1} & \text{si } i = 0 \text{ et } j \neq 0 \\ w_{i-1,0} & \text{si } i \neq 0 \text{ et } j = 0 \\ \max(w_{i,j-1}, w_{i-1,j}, w_{i-1,j-1}) & \text{si } i \neq 0 \text{ et } j \neq 0 \end{cases}$$

On peut calculer $w_{n-1,p-1}$ par programmation dynamique.

On adopte l'approche ascendante de la programmation dynamique : on remplit la matrice $W = (w_{i,j})_{i,j}$ soit par lignes croissantes (et colonnes croissantes), soit par colonnes croissantes (et lignes croissantes), soit par anti-diagonales, *i.e* par $i+j$ croissant.

Exo : code

Complexité : $\mathcal{O}(np)$, en espace aussi, mais on peut optimiser l'espace requis car il suffit de connaître une ligne ou une colonne pour remplir la suivante : $\mathcal{O}(\min(n, p))$ en espace.

Remarque : on n'a calculé que le poids maximal, pas un chemin réalisant ce poids.

Deux possibilités :

- (1) Lors du remplissage de la matrice, on détermine en plus de $w_{i,j}$ un prédécesseur de (i, j) sur un chemin de poids maximal de $(0, 0)$ à (i, j) . On détermine un tel prédécesseur en gardant un couple qui réalise le maximum dans la formule de récurrence.
- (2) On détermine la matrice W puis on retrouve un chemin de poids maximal en partant de la case $(n-1, p-1)$, en déterminant un prédécesseur à l'aide des valeurs de N puis en calculant récursivement un chemin vers ce prédécesseur.

3.2.3 Exemple : Produit de matrices

On cherche à minimiser le nombre de multiplications nécessaires au calcul du produit de n matrices.

Si $A \in \mathcal{M}_{n,p}(K)$, $B \in \mathcal{M}_{p,q}(K)$, on peut calculer AB à l'aide de npq multiplications



grâce à la définition :

$$(AB)_{i,j} = \sum_{k=1}^p a_{i,k} b_{k,j}$$

Si A, B, C sont de tailles respectives 2×5 , 5×8 , 8×6 , on peut calculer ABC de deux façons :

- $(AB)C$: 176 multiplications ;
- $A(BC)$: 300 multiplications.

On veut donc trouver un parenthésage optimal du produit

$$\prod_{k=0}^{n-1} A_k$$

où $\forall k \in \llbracket 0 ; n-1 \rrbracket$, $A_k \in \mathcal{M}_{t_k, t_{k+1}}(K)$.

Ce problème vérifie la propriété de sous-problème optimal : si

$$\left(\prod_{k=0}^i A_k \right) \left(\prod_{k=i+1}^{n-1} A_k \right)$$

est un parenthésage optimal, on a bien des parenthésages optimaux des produits

$$\prod_{k=0}^i A_k \quad \text{et} \quad \prod_{k=i+1}^{n-1} A_k$$

On observe qu'il faudra résoudre les sous-problèmes suivants : optimisation du parenthésage du produit

$$P_{i,j} = \prod_{k=i}^j A_k$$

$\forall (i, j) \in \llbracket 0 ; n-1 \rrbracket^2$. On note $m_{i,j}$ le nombre minimal de multiplications nécessaires au calcul de $P_{i,j}$.

Si $\underbrace{P_{i,k}}_{\in \mathcal{M}_{t_i, t_{k+1}}(K)} \times \underbrace{P_{k+1,j}}_{\in \mathcal{M}_{t_{k+1}, t_{j+1}}(K)}$ est un parenthésage optimal, alors

$$m_{i,j} = m_{i,k} + m_{k+1,j} + t_i t_{k+1} t_{j+1}$$

On trouve alors la relation de récurrence

$$m_{i,j} = \begin{cases} 0 & \text{si } i = j \\ \min_{k \in \llbracket i ; j-1 \rrbracket} (m_{i,k} + m_{k+1,j} + t_i t_{k+1} t_{j+1}) & \text{si } i < j \end{cases}$$

Comment remplir la matrice $M = (m_{i,j})_{i,j}$?

$$\begin{pmatrix} 0 & & & & \\ & \ddots & \leftarrow & \bullet & \\ & & \ddots & \downarrow & \\ & & & \ddots & \\ & & & & 0 \end{pmatrix}$$

On peut remplir la matrice par diagonales, *i.e* par $j - i$ croissant.

Exo code.

Complexité : $\mathcal{O}(n^3)$ en temps, et $\mathcal{O}(n^2)$ en espace.

Comment représenter un parenthésage optimal $\Pi_{i,j}$ de $P_{i,j}$?

Si le parenthésage s'écrit $P_{i,k}P_{k+1,j}$, il suffit de retenir k et les parenthésages $\Pi_{i,k}$ et $\Pi_{k+1,j}$

