Chapitre 2
Programmation dynamique (partie 1)

Table des matières

1 Exemple : le produit matriciel

1.1 Produit de deux matrices

Soit $A \in \mathscr{M}_{n,p}(\mathbb{R})$ et $B \in \mathscr{M}_{q,r}(\mathbb{R})$. Rappelons que le produit AB n'est défini que si $p = q$. Dans ce cas, $C = AB \in \mathscr{M}_{n,r}(\mathbb{R})$ et pour tout $(i,j) \in [\![1\,;n]\!] \times [\![1\,;r]\!]$, $C_{i,j} = \sum_{k=1}^{p} A_{i,k}B_{k,j}$. Il y a alors p multiplications pour calculer un coefficient $C_{i,j}$ et il y nr coefficients à calculer. Ainsi, il faut npr multiplications [1] pour calculer la matrice AB.

1.2 Produit de trois matrices

Considérons maintenant $A \in \mathscr{M}_{n,p}(\mathbb{R})$, $B \in \mathscr{M}_{p,q}(\mathbb{R})$ et $C \in \mathscr{M}_{q,r}(\mathbb{R})$ de sorte qu'on puisse effectuer le produit ABC. Par associativité du produit matriciel, $(AB)C = A(BC)$. Il y a donc deux façons de calculer ABC :

- Le calcul de AB demande npq multiplications, puis le calcul de $(AB)C$ demande nqr multiplications. Ainsi, au total, il y aura $nq(p+r)$, multiplications.
- Le calcul de BC demande pqr multiplications, puis le calcul de $A(BC)$ demande npr multiplications. Ainsi, au total il y aura $pr(n+q)$, multiplications.

Or, $nq(p+r)$ et $pr(n+q)$ n'ont aucune raison d'être égaux. Par exemple si $n = 5$, $p = 10$, $q = 2$ et $r = 20$, alors le produit $(AB)C$ donne 300 multiplications. Tandis que $A(BC)$ donne 1400 multiplications.

1.3 Généralisation à un produit de n matrices

Soient $n+1$ matrices $A_i \in \mathscr{M}_{p_i,p_{i+1}}(\mathbb{R})$ pour $i \in [\![0\,;n]\!]$, de sorte qu'on puisse calculer $A_0 \times A_1 \times A_2 \times \ldots \times A_n \in \mathscr{M}_{p_0,p_{n+1}}(\mathbb{R})$. Attention, ici il y a $n+1$ matrices et non n, n représente le nombre de produits à effectuer. Commençons par compter le nombre de parenthésages possibles pour n petit :

n	liste des parenthèses	$C(n)$: nombre de parenthésages possibles
0	A_0	1
1	A_0A_1	1
2	$A_0(A_1A_2)$, $(A_0A_1)A_2$	2
3	$A_0((A_1A_2)A_3)$, $A_0(A_1(A_2A_3))$, $(A_0A_1)(A_2A_3)$, $((A_0A_1)A_2)A_3$, $(A_0(A_1A_2))A_3$	5

À l'aide d'un produit de Cauchy, on peut montrer que $C(n) = \frac{1}{n}\binom{2n-2}{n-1}$. On voit ainsi, que $C(n) \sim K4^n/n^{\frac{3}{2}}$. Ainsi, chercher, parmi tous les parenthésages, celui qui prend le moins de calculs, est illusoire. Cherchons plus efficace !

1.4 Méthode de programmation dynamique

Pour calculer $A_0 \times A_1 \times \ldots \times A_n$ de façon optimale, généralisons et cherchons à calculer $A_i \times A_{i+1} \times \ldots \times A_j$ de façon optimale pour $(i,j) \in [\![0\,;n]\!]^2$ avec $i \leqslant j$. Lorsque l'on calcule $A_i \times A_{i+1} \times \ldots \times A_j$ avec des parenthèses de façon à obtenir le moins de multiplications de nombres possibles, ce calcul est de la forme $(A_i...A_k) \times (A_{k+1}...A_j)$ avec $k \in [\![i\,;j-1]\!]$. Pour réaliser ce calcul, il faudra donc faire trois choses :

1. Calculer $B = A_i...A_k \in \mathscr{M}_{p_i,p_{k+1}}(\mathbb{K})$ de façon optimale.
2. Calculer $C = A_{k+1}...A_j \in \mathscr{M}_{p_{k+1},p_{j+1}}(\mathbb{K})$ de façon optimale.
3. Effectuer le produit BC.

Le produit BC prend $p_ip_{k+1}p_{j+1}$ multiplications. Le problème, c'est qu'on ne connaît ni k ni le nombre de façons de calculer B et C de façon optimale. Après tout, si on ne sait pas calculer $A_i \ldots A_j$ de façon optimale, pourquoi serions nous capable de calculer $A_i \ldots A_k$ et $A_{k+1} \ldots A_j$ de façon optimale ? Mais l'on peut remarquer que dans le calcul de B et de C, il y a moins de matrices que dans le produit $A_i \times \ldots \times A_j$. Ainsi, on pourrait travailler par récursivité. Notons $c(i,j)$ le nombre de multiplications nécessaires pour calculer $A_i \times \ldots \times A_j$ de façon optimale. On obtient donc

$$c(i,j) = c(i,k) + c(k+1,j) + p_ip_{k+1}p_{j+1}$$

Mais on ne connaît toujours pas k. Seulement, on cherche, à faire le moins de multiplications possibles, ainsi on a intérêt à choisir k qui minimise le coût total. Ainsi :

$$c(i,j) = \min_{k \in [\![i\,;j-1]\!]} c(i,k) + c(k+1,j) + p_ip_{k+1}p_{j+1}$$

On peut donc calculer, les $c(i,j)$ par récursivité, en constatant que $c(i,i) = 0$. Cela conduit au code Python suivant :

1. On néglige souvent les additions par rapport aux multiplications ce genre de calcul de complexité.

```
def cout(i,j):
    if i == j:
        return 0
    return min(cout(i,k)+cout(k+1,j)+P[i]*P[k+1]*P[j+1] for k in range(i,j))
```

Où `P` est la liste des tailles de lignes/colonnes des matrices A_i : ainsi pour tout $i \in [\![0 ; n+1]\!]$, `P[i]=`p_i. Alors avec `n=len(P)-2`, `c(0,n)` donne le résultat voulu. Cependant, la fonction `min` ne sera pas accepté aux concours [2], on va donc refaire le code en calculant le minimum :

```
def cout(i,j):
    if i == j:#
        return 0
    else:#calcul du minimum
        m = np.inf
        for k in range(i,j):
            A = cout(i,k) + cout(k+1,j) + P[i]*P[k+1]*P[j+1]
            if A < m:
                m = A
        return m
```

Si on teste avec $n = 20$, on voit que le temps devient anormalement lent. Essayons de comprendre pourquoi. La figure 1 montre les différents coefficients $c(i,j)$ qui vont être nécessaires pour calculer $c(0,4)$ (le coût optimal pour calculer $A_0A_1A_2A_3A_4$). On s'aperçoit que certains coefficients sont calculés plusieurs fois. Or, si vous avez une fonction Python `f` et que vous lui demandez de calculer `f(10)` cinquante fois, Python recalculera le résultat de la fonction cinquante fois [3]. Il faut donc lui demander de stocker chaque résultat et de s'y référer.

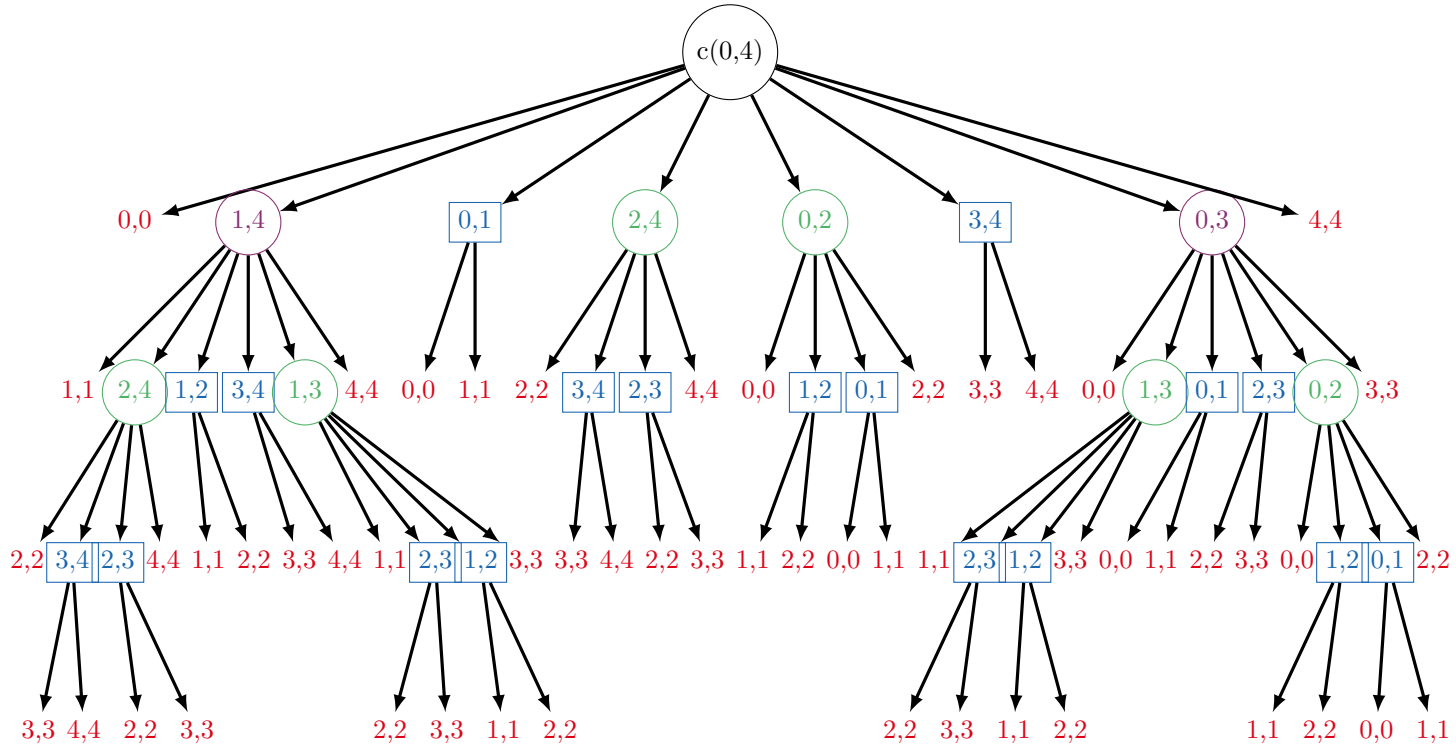

FIGURE 1 – Dépendance des calculs par récursivité.

2. En effet, savoir programmer un minimum est un minimum aux concours.
3. On aurait pu imaginer qu'il stocke le résultat en mémoire pour le réutiliser plus tard, mais ce n'est pas le cas, car cela pourrait causer des problèmes de mémoire si Python stockait tout les résultats des calculs qu'il effectue.

1.5 Amélioration par mémoïsation

La mémoïsation consiste donc à stocker chaque calcul de la fonction. Ici, on utilisera un dictionnaire, à chaque fois qu'un $c(i,j)$ est calculé, on insérera la clé (i,j) dans un dictionnaire (vide au départ) et dont la valeur sera le $c(i,j)$ calculé. Lorsque la fonction doit retourner $c(i,j)$ au lieu de refaire le calcul, la fonction vérifiera d'abord si (i,j) est une clé [4], si (i,j) est une clé, alors la fonction renvoie la valeur de la clé, sinon elle procède par récursivité. Ainsi, pour faire le calcul de $c(0,n)$, on aura besoin d'un certain nombre de coefficients $c(i,j)$, mais le calcul de $c(i,j)$ ne sera fait qu'une seule fois. Dès lors, si on dessine la dépendance des calculs des $c(i,j)$ on s'aperçoit que l'arbre perd beaucoup de feuilles [5].

```
Dc={}
def coutavecmemoisation(i,j):
    if (i,j) in Dc:
        return Dc[(i,j)]
    elif i == j:
        Dc[(i,j)] = 0
        return 0
    else:
        m = np.inf
        for k in range(i,j):
            A = coutavecmemoisation(i,k) + coutavecmemoisation(k+1,j) + P[i]*P[k+1]*P[j+1]
            if A < m:
                m = A
        Dc[(i,j)] = m
        return m
```

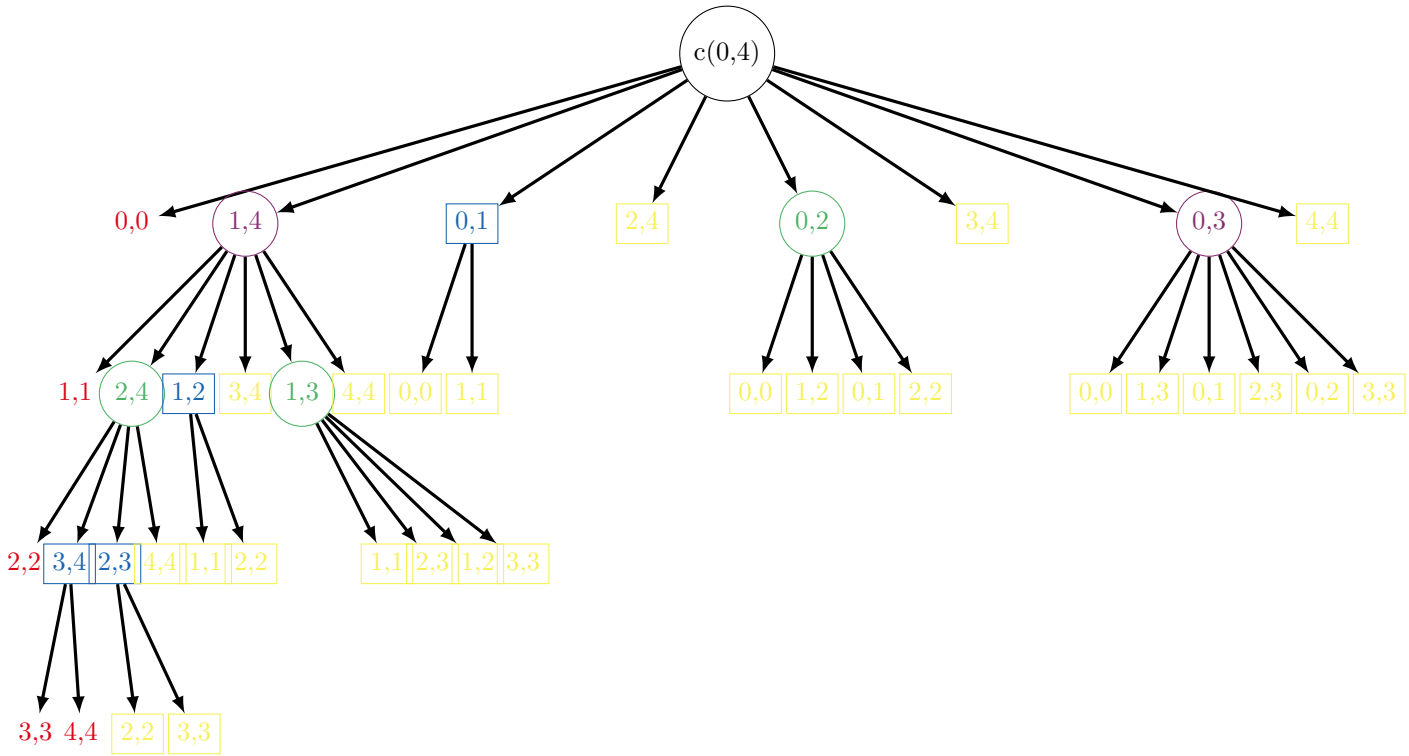

FIGURE 2 – Dépendance des calculs par récursivité. La mémoïsation a diminué le nombre d'appels de la fonctions. En jaune, les calculs qui ont déjà été faits et qui ne seront donc pas refaits.

Il est possible de compter le nombre d'appels récursifs des fonctions avec mémoïsation et sans mémoïsation en fonction du nombre de matrices. La figure suivante compare les deux approches :
Ainsi, le temps de calcul sera largement plus raisonnable. Il y a cependant un prix à payer, on gagne du temps mais on perd en mémoire, car on occupe plus de mémoire pour stocker les différents résultats.

4. ce qui est rapide à faire rappelons-le
5. Autumn is coming !

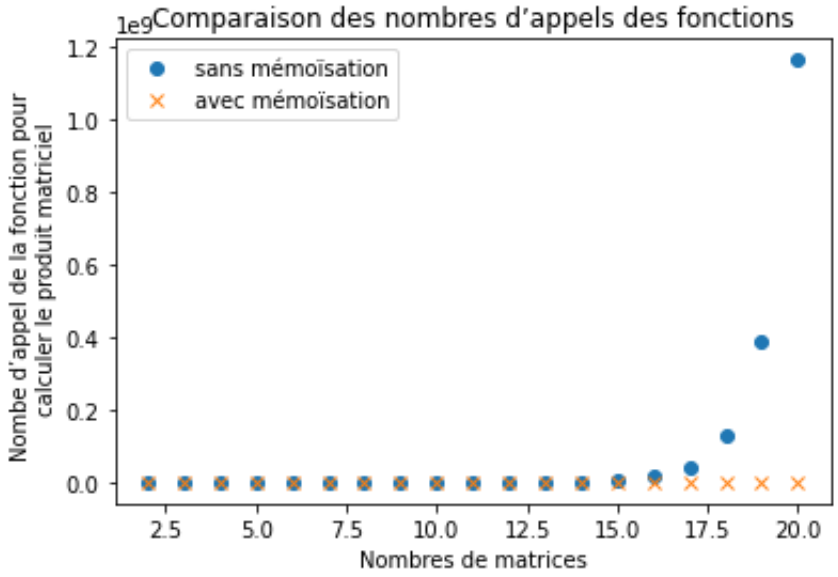

FIGURE 3 – Nombres d'appels des fonctions avec ou sans mémoïsation en fonction du nombre de matrices.

1.6 Reconstitution de la solution optimale

Il est certes intéressant de connaître le nombre de multiplications optimal qu'il va falloir faire. Mais le mieux serait quand même de savoir comment mettre les parenthèses pour effectuer le produit correspondant à ce nombre de multiplications optimal. Il faut bien comprendre que, lorsque l'on a précédemment minimiser la valeur de `c(i,j)` c'était en minimisant une quantité qui dépendait de k, on a retenu cette quantité minimale, mais pas le k correspondant. On va donc modifier la fonction, pour qu'elle renvoie maintenant un couple $(c(i,j),k)$ ou k est l'entier qui a atteint le minimum. Ainsi, si on veut calculer $A_iA_{i+1}\dots A_j$ non seulement, on sera que $c(i,j)$ est le nombre de multiplications optimale. Mais qu'en plus, que pour atteindre cet optimum, il faut faire $(A_iA_{i+1}...A_k)\times(A_{k+1}\times\dots A_j)$.

```
def coutavecmemoisationP(i,j):
    if (i,j) not in Dc:
        if i == j:
            Dc[(i,j)] = (0,i)
        else:
            m = np.inf
            for k in range(i,j):
                A = coutavecmemoisationP(i,k)[0] + coutavecmemoisationP(k+1,j)[0] + P[i]*P[k+1]*P[j+1]
                if A < m:
                    m = A
                    k0 = k
            Dc[(i,j)] = (m,k0)
    return Dc[(i,j)]
```

Ainsi, on va pouvoir créer une fonction récursive qui va afficher le parenthésage optimal :

```
def par(i,j):
    if i == j:
        return("A"+str(i))
    k = coutavecmemoisationP(i,j)[1]
    return "("+par(i,k)+"x"+par(k+1,j)+")"
```

1.7 Terminaison/correction/complexité

2 Définition de la programmation dynamique

Fort de cet exemple, nous pouvons donner une définition de la programmation dynamique : il s'agit de trouver un résultat d'optimisation en découpant ce problème en sous-problèmes qu'il va falloir eux-même optimiser. Ces sous-problèmes vont

se chevaucher, c'est-à-dire qu'un même sous-problème va se retrouver plusieurs fois. Il va falloir aussi comprendre comment la résolution des sous-problèmes amène à résoudre le problème original.

En général, les problèmes de programmation dynamique fonctionne en deux étapes : on cherche d'abord à calculer le coût optimal, puis on modifie le programme déjà crée pour rajouter la construction de la solution optimale.