TP8 : Programmation dynamique

Thomas Bourgeat Paul Melotti (paul.melotti@ens.fr)

6 et 13 février 2014

1 Multiplications matricielles optimales

Soit $A_1, \dots, A_n$ des matrices rectangulaires, telles que

$$A_1 \in \mathcal{M}_{p_0,p_1}, A_2 \in \mathcal{M}_{p_1,p_2}, \dots, A_n \in \mathcal{M}_{p_{n-1},p_n}.$$

On cherche à calculer le produit $A_1 \times \cdots \times A_n$. On dispose du tableau $p = [|p_0; \dots p_n|]$ des tailles successives.

Question.1 Combien de multiplications faut-il effectuer pour calculer le produit AB où $A \in \mathcal{M}_{p,q}$ et $B \in \mathcal{M}_{q,r}$?

Par exemple, si $p = [|10; 100; 5; 50|]$, combien faut-il d'opérations pour calculer le produit si on le parenthèse en $(A_1 \times A_2) \times A_3$ et en $A_1 \times (A_2 \times A_3)$?

On constate que l'ordre dans lequel on effectue les multiplications est important. L'objectif de cette partie est de trouver le parenthésage optimal pour calculer le produit.

Si $i \leq j$, on note m_{ij} le nombre minimal de multiplications scalaires pour calculer le produit $A_i \dots A_j$, et M la matrice des m_{ij} (qui est triangulaire supérieure de diagonale nulle).

Le parenthésage optimal de ce calcul sépare le produit en $A_i \dots A_j = (A_i \dots A_k)(A_{k+1} \dots A_j)$ pour un certain indice de coupure $k \in [i; j-1]$. On note s_{ij} ce k, et S la matrice des s_{ij}.

Question.2 Montrer que

$$m_{ij} = \begin{cases} 0 & \text{si } i = j \\ \min_{i \leq k < j}(m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j) & \text{si } i < j. \end{cases}$$

Cette formule permet de remplir la matrice M intelligemment si on calcule les coefficients dans un certain ordre, lequel ?

Question.3 Écrire une fonction `couts : int vect -> int vect vect` qui étant donné le vecteur `p` renvoie la matrice `M`. Quelle est sa complexité ?

Question.4 Modifier la fonction précédente en une fonction `coupes : int vect -> int vect vect` qui étant donné `p` renvoie la matrice `S`.

Question.5 Écrire une fonction `parenthesages : int vect -> unit` qui étant donné `p` imprime le parenthésage optimal. Par exemple :

```
#parenthesages [|3;2;1;4;2;2|] ;;
((1)(2))(((3)(4))(5))- : unit = ()
```

Question.6 (*) Combien y a-t-il de façons de parenthéser le produit $A_1 \dots A_n$? Quelle serait la complexité d'un algorithme qui énumèrerait tous les parenthésages et calculerait le coût de chacun ?

2 Un peu de réflexion autour de la programmation dynamique

L'idée de la programmation dynamique est de pouvoir résoudre un problème d'*optimisation* en commençant par résoudre des sous-problèmes et en recombinant les solutions de ces sous-problèmes.

Pour que cela soit possible, il faut que le problème ait une structure particulière. Par exemple s'il vérifie la propriété de sous-structure optimale : lorsqu'on a une solution optimale à notre problème, cette solution contient des solutions optimales à des sous-problèmes. Par exemple, pour la section précèdente, trouver un bon parenthèsage pour $A_1 \dots . A_n$ nous donne un bon parenthèsage pour un couple de $A_1 \dots . A_i$ et $A_{i+1} \dots . A_n$.

Autrement dit, la solution au gros problème donne des solutions à des petits problèmes dont il est composé.

Souvent, les stratégies de programmation dynamique consistent à grossir un peu le problème : dans la section précédente, on a trouvé des parenthésages optimaux pour plein de couples (i, j) inutiles en pratique.

Question.0 (orale) Parfois les étudiants confondent récursivité et programmation dynamique. Pensez-y, pourquoi est-ce que ce n'est pas la même chose, pourquoi est-ce qu'il y pourrait y avoir confusion ?

Question.1 Avez-vous des idées de problèmes qui pourraient se traiter par programmation dynamique ? De problèmes ne s'y prêtant pas ?

2.1 Dynamique contre glouton

On s'intéresse au problème du rendu de monnaie : on dispose de pièces (on en a autant qu'on veut !) de valeurs suivantes : $\{p_1, \dots, p_n\}$, où les p_i sont triées par ordre croissant. On cherche le nombre minimum de pièces qu'il faut pour payer une somme s.

Une première idée est d'utiliser un algorithme *glouton* : on essaie de payer s en mettant la plus grande pièce possible p_i, puis on est ramené à payer $s - p_i$ et on recommence. De manière générale, un algorithme glouton est un algorithme qui tente à chaque étape d'utiliser un optimum local (*i.e.* de manger le plus gros morceau possible).

Question.2 Écrire une fonction `glouton : int list -> int -> int` qui tente de résoudre le problème du rendu de monnaie par l'algorithme glouton (on reçoit en entrée la liste des pièces disponibles et la somme à payer, et on renvoie le nombre de pièces utilisées pour payer). Quelle est sa complexité ?

Question.3 La fonction précédente est-elle optimale ? Par exemple, si je dois payer 6 et que les pièces disponibles sont 1, 3 et 4 ?

Question.4 Résoudre le problème de façon optimale par programmation dynamique. Quelle est la complexité de votre fonction ?
Indication : on pourra calculer le nombre de pièces optimal pour toutes les sommes $i \leq s$.

2.2 Facultatif - La revanche du glouton

On s'intéresse au problème de la gestion d'une salle : n personnes veulent organiser des événements dans notre salle avec des horaires de début d_i et de fin f_i pour chaque événement. Mais certains horaires sont incompatibles. On souhaite maximiser le nombre d'événements dans notre salle (et non pas le temps d'occupation).

Question.5 On propose plusieurs algorithmes gloutons pour répondre au problème. Dites si ces idées donnent une solution optimale ; si oui, donnez une preuve, et si non, donnez un contre-exemple [1].

a) Mettre à chaque fois l'événement le plus court parmi les événements compatibles.

b) Mettre à chaque fois l'événement qui commence le plus tôt parmi les événements compatibles.

c) Mettre à chaque fois l'événement qui finit le plus tôt parmi les événements compatibles.

1. Conseil : cherchez d'abord des contre-exemples

Question.6 Écrire une fonction `salle : int vect -> int vect -> int` qui étant donné les tableaux des d_i et des f_i (les i étant dans un ordre quelconque) donne le nombre maximal d'événements qu'on peut organiser dans notre salle, en utilisant l'algorithme glouton approprié. Quelle est sa complexité ?

Question.7 Voyez-vous comment résoudre ce problème par programmation dynamique ? Écrivez sur papier une idée d'algorithme. Quelle serait sa complexité ?

3 Quelques exercices faisables en dynamique

3.1 Fibonacci (encore...)

Dans le cas du calcul du n-ième terme de la suite de Fibonacci, en quoi consisterait une approche dynamique ? Coder cette fonction.

3.2 Problème du sac à dos

On dispose de n objets de poids $p_1, ..., p_n$ et de valeur (pécuniaire) $v_1, ..., v_n$. On souhaite remplir un sac à dos de manière à transporter la plus grande valeur possible, mais il ne peut contenir qu'un poids inférieur à P. Trouver un algorithme qui donne la valeur maximale qu'on peut transporter.

Indication : On pourra s'intéresser à $V_{i,p}$ la valeur maximale qu'on peut transporter si on n'a que les i premiers objets et un sac à dos de contenance $p \leq P$.

3.3 Problème de gain

Énoncé de Louis Jachiet :
Après avoir voyagé dans le futur puis être revenu à votre époque, vous disposez d'un tableau qui vous dit ce que vaut, chaque mois, le cours de l'or. Comme vous vou- lez à tout prix vous enrichir, mais en faisant le moins d'efforts possible, vous décidez de regarder quel est le profit maximal que vous puissiez faire. Le cours de l'or est présenté sous la forme d'une liste d'entiers et vous devez donner le gain maximal réalisable. Vous ne pouvez faire qu'un achat/vente car sinon vous modifieriez trop le cours du temps et donc cela changerait les cours (vous pourriez y perdre) mais cela risquerait aussi de modifier le continuum espace/temps lui même !

Vous cherchez donc le gain maximum, qui s'exprime par

$$G = \max_{0 \leq i \leq j \leq n-1} (v_j - v_i).$$

Question.1 G est-il toujours égal à la différence entre le maximum et le minimum du tableau ?

Question.2 Écrire une fonction efficace qui détermine G.
Indication : $g_i = \max_{0 \leq k \leq i}(v_i - v_k)$

Question.3 Donner un algorithme qui détermine de manière efficace la plus grande somme formée par des éléments consécutifs d'un tableau de nombres.

3.4 Sous-chaîne commune

Écrire un algorithme qui trouve la plus longue sous-séquence commune à deux chaînes de caractères.

3.5 (**) Points fixes d'une itérée de fonction affine par morceaux

Soit m un entier, et f une fonction de $[0; m]$ dans $[0; m]$ telle que :
– Pour tout entier $i \leq m$, $f(i)$ est un entier noté f_i ;
– Pour tout entier $i < m$, f est affine sur le segment $[i; i+1]$.
Donner une algorithme qui, étant donné le tableau des f_i et un entier k, détermine le nombre de points fixes de f^k (l'itérée k-ième de f). Votre algorithme devra être au plus linéaire en k.