Programmation dynamique

La programmation dynamique s'applique le plus souvent aux problèmes d'optimisation pour lesquels on doit faire un ensemble de choix pour arriver à une solution optimale. À mesure que de nouvelles options sont choisies, des sous-problèmes de la même forme apparaissent souvent. La programmation dynamique est efficace lorsqu'un sous-problème donné peut apparaître pour plusieurs suites d'options différentes. Le principe est de stocker la solution à chaque sous-problème au cas où il réapparaîtrait. Dans ce sujet, nous étudions deux tels exemples.

1 Plus longue sous-suite commune

On considère une suite finie $x = (x_0, \ldots, x_{m-1})$ formée de m éléments d'un ensemble E. Une *sous-suite* de x de longueur k est une suite x' obtenue à partir de x en supprimant $m-k$ éléments tout en conservant l'ordre : $x' = (x_{\sigma(0)}, \ldots, x_{\sigma(k-1)})$ avec $0 \leq \sigma(0) < \cdots < \sigma(k-1) < m$. On dit que z est une sous-suite commune de x et y si z est une sous-suite de x et une sous-suite de y.
Nous représenterons en Caml les suites finies par des tableaux. Nous cherchons maintenant à trouver une plus longue sous-suite commune à deux suites $x = (x_0, \ldots, x_{m-1})$ et $y = (y_0, \ldots, y_{n-1})$. Une méthode naïve consisterait à construire l'ensemble des sous-listes de x et de y, puis à calculer leur intersection.

▶ **Question 1** *Évaluez la complexité, en fonction de la taille des listes x et y de cette méthode.*

Dans un premier temps, nous nous limitons au problème du calcul de la *longueur* d'une plus longue sous-suite commune de deux suites x et y. On note $\ell(i,j)$ la longueur de la plus longue sous suite commune des suites $(x_0, \ldots, x_{i-1})$ et $(y_0, \ldots, y_{j-1})$. La longueur recherchée est donnée par $\ell(m,n)$ (où m et n sont les longueurs respectives de x et y). De plus, ℓ vérifie la relation de récurrence suivante :

$$\ell(i,j) = \max(\ell(i-1,j-1) + \delta(x_{i-1}, y_{j-1}), \ell(i,j-1),\ \ell(i-1,j))$$

où $\delta(x_i, y_j) = 1$ si $x_i = y_j$ et 0 sinon.
Il est facile de vérifier que si on utilise cette relation pour écrire une fonction récursive calculant la valeur de $\ell(i,j)$, on obtiendrait à nouveau une fonction dont le coût serait exponentiel en la taille de l'entrée : l'algorithme récursif rencontre en effet chaque sous-problème de nombreuses fois dans différentes branches de l'arbre récursif, sans réutiliser toutefois les calculs déja effectués. La programmation dynamique contourne ce problème en stockant les résultats intermédiaires dans un tableau à deux dimensions auxilliaire : la case (i,j) de cette matrice contenant la longueur d'une plus longue sous-liste commune de $(x_0, \ldots, x_{i-1})$ et $(y_0, \ldots, y_{j-1})$, c'est-à-dire $\ell(i,j)$.

▶ **Question 2** *Expliquez comment on peut remplir de proche le tableau auxilliaire.*

▶ **Question 3** *Déduisez-en une fonction qui permette de calculer la longueur de la plus longue sous-suite commune à deux suites. Votre fonction aura une complexité dominée par le produit des longueurs des deux listes passées en argument.*

```
val subseq_length: 'a vect -> 'a vect -> int
```

On souhaite maintenant obtenir une des sous-suites communes de longueur maximale de deux suites. Pour cela, remarquons que la valeur de chaque case du tableau auxilliaire est obtenue à partir de celle de l'une de ses trois voisines : celle située au dessus, celle située à gauche ou celle située en diagonale en haut à gauche. En conservant (dans un second tableau) pour chaque case une marque indiquant à partir de quelle voisine sa valeur a été calculée, il est possible à la fin de l'algorithme de reconstituer une plus longue sous-suite commune en parcourant le tableau obtenu en suivant les directions indiquées par les marques comme illustré sur la figure suivante :

	j	0	1	2	3
i		y_{j-1}	a	b	c
0	x_{i-1}	0	0	0	0
1	a	0	↖ 1	← 1	← 1
2	a	0	↖ 1	↑ 1	↑ 1
3	c	0	↑ 1	↑ 1	↖ 2
4	b	0	↑ 1	↖ 2	↑ 2

Pour représenter les marques, vous pouvez définir en Caml le type suivant :

```
type mark = Vert | Horiz | Diag;;
```

▶ **Question 4** *Écrivez une fonction* subseq *qui retourne une des plus longues sous-suites commune de deux suites.*

val subseq: 'a vect −> 'a vect −> 'a vect

2 Multiplication d'une suite de matrices

Notre deuxième exemple de programmation dynamique est un algorithme qui résoud le problème de la multiplication d'une suite de matrices. On considère une suite de n matrices d'entiers $(M_0, M_1, \ldots, M_{n-1})$ à multiplier, i.e. on souhaite calculer le produit matriciel $M_0 M_1 \cdots M_{n-1}$. Il est possible d'évaluer ce produit en utilisant l'algorithme classique permettant de multiplier deux matrices, après avoir placé des parenthèses pour préciser l'ordre dans lequel les produits binaires doivent être effectués. La multiplication des matrices étant associative, tous les parenthésages aboutissent à une même valeur du produit. Cependant, la manière dont une suite de matrices est parenthésée peut avoir un impact crucial sur le coût d'évaluation du produit.

▶ **Question 5** *Écrivez une fonction qui calcule le produit de deux matrices d'entiers. (Dans le cas où la taille des matrices n'est pas compatible, vous lèverez une exception.) Précisez le coût (en nombre de multiplication d'entiers) du produit de deux matrices en fonction de leur taille.*

val prod: int vect vect −> int vect vect −> int vect vect

▶ **Question 6** *Donnez un exemple de dimensions pour trois matrices M_0, M_1 et M_2 tel que le coût du calcul de $(M_0 M_1)M_2$ soit différent de celui de $M_0(M_1 M_2)$. Montrez que le rapport entre ces deux coûts peut être arbitrairement grand.*

Pour le problème qui nous intéresse, on peut représenter la suite de matrices $(M_0, \ldots, M_{n-1})$ par un tableau de $n+1$ entiers d tels que, pour tout i, la matrice M_i a pour dimensions $(d.(i), d.(i+1))$. On note également, pour $i \leq j$, $M_{i,j}$ le produit $M_i M_{i+1} \cdots M_j$ et $p_{i,j}$ le nombre minimum de multiplications d'entiers nécessaires pour le calcul de ce produit.

▶ **Question 7** *Montrez que si $i < j$ alors $p_{i,j}$ est égal au minimum de l'ensemble $\{p_{i,k} + p_{k+1,j} + d.(i) \times d.(k+1) \times d.(j+1) \mid i \leq k < j\}$.*

▶ **Question 8** *Écrivez une fonction* min_fun *prenant pour arguments une fonction f de type* int −> int *et deux entiers i_1 et i_2 (tels que $i_1 \leq i_2$). Votre fonction calculera l'entier i compris entre i_1 et i_2 tel que $f(i)$ soit minimal.*

val min_fun: (int −> int) −> int −> int −> int

▶ **Question 9** *Déduisez-en une fonction qui calcule le nombre minimal de multiplications entières à effectuer pour calculer le produit d'une suite de matrices. (Votre fonction prendra pour argument le tableau des dimensions d.)*

val optprod: int vect −> int

La fonction que vous venez d'écrire a probablement un temps d'exécution exponentiel, ce n'est pas meilleur que la méthode directe consistant à tester chaque manière de parenthéser le produit : l'algorithme récursif rencontre en effet chaque sous-problème de nombreuses fois dans différentes branches de l'arbre récursifs, sans réutiliser toutefois les calculs déja effectués.
L'approche par *programmation dynamique* consiste à calculer incrémentalement les valeurs de $p_{i,j}$ dans l'ordre des $j - i$ croissants et à stocker les valeurs obtenues dans un tableau à deux dimensions. Ainsi, le contenu de chaque case du tableau n'est évalué qu'une seule fois et son calcul ne fait appel qu'à des cases déjà calculées. Voici le tableau obtenu pour une suite de quatre matrices de dimensions 2×4, 4×3, 3×1 et 1×5 :

$p_{i,j}$	0	1	2	3
0	0	24	20 $(k=1)$	30 $(k=2)$
1		0	12	32 $(k=2)$
2			0	15
3				0

▶ **Question 10** *Écrivez une fonction* optprod_dyn *qui calcule le nombre minimal de multiplications entières à effectuer pour calculer le produit d'une liste de matrices en utilisant cette méthode.*

val optprod_dyn: int vect −> int

▶ **Question 11** *Évaluez la complexité de la fonction* optprod_dyn.

On souhaite maintenant obtenir la forme du parenthésage qui permet d'atteindre le minimum. Il suffit pour cela d'utiliser un tableau à deux dimensions auxilliaire s tel que $s_{i,j}$ contienne la valeur de k pour laquelle le minimum défini à la question 7 a été atteint. Pour représenter un tel parenthésage, on peut utiliser un arbre binaire dont les feuilles sont étiquetées par les numéros des matrices :

```
type tree =
    Matrix of int
  | Product of tree * tree
;;
```

▶ **Question 12** *Déduisez-en une fonction qui construit l'arbre représentant le meilleur parenthésage pour calculer le produit d'une liste de matrices.*

val optprod_tree: int vect −> tree

Programmation dynamique

Un corrigé

▶ **Question 1** Evaluation de la complexité, dans le pire des cas :
On commence par supposer que tous les éléments de x sont distincts. Notons $n(k,m)$ le nombre de sous-suites de longueur k d'un mot de longueur n. On a $n(k,m) = n(k-1,m-1)+n(k,m-1)$. Comme on a les mêmes conditions initiales (une sous-suite de longueur 0, ...), on montre que l'on a $n(k,m) = \frac{k!}{((k-m)!m!)}$
Bref, c'est gros, exponentiel et donc infaisable en pratique.

▶ **Question 2** $l(i,j)$ fait appel à $l(i-1,j-1)$, $l(i,j-1)$ et $l(i-1,j)$. Si on veut remplir $l(i,j)$, il faut avoir calculé ces trois valeurs, donc calculer les $l(i,j)$ par $i+j$ croissant.

▶ **Question 3**

```
let d = fun x y i j -> if x.(i-1)=y.(j-1) then 1 else 0;;

let subseq_length = fun x y ->
  let m = vect_length x and n = vect_length y in
  let l = make_matrix (m+1) (n+1) 0 in
    for i = 1 to m do
      for j = 1 to n do
 l.(i).(j) <- max
           (l.(i-1).(j-1) + (d x y i j))
           (max l.(i).(j-1) l.(i-1).(j))
      done
    done;
    l.(m).(n);;

let t1 = [|'a';'b';'c'|];;
let t2 = [|'c';'b';'c'|];;
subseq_length t1 t2;;
```

▶ **Question 4** Notre fonction comporte deux parties. La première remplit incrémentalement les matrices s et t. Elle est analogue à la fonction subseq_length. La deuxième partie parcours le la matrice t de manière à retrouver construire une des plus longues sous-séquences communes.

```
let maxi = fun x y l i j ->
  if x.(i-1) = y.(j-1) then ((fst l.(i-1).(j-1))+1, Diag)
  else if (fst l.(i-1).(j)) > (fst l.(i).(j-1)) then
    ((fst l.(i-1).(j)), Vert)
    else ((fst l.(i).(j-1)), Horiz);;

let subseq = fun x y ->
(* on construit la meme matrice, mais en memorisant le chemin *)
  let m = vect_length x and n = vect_length y in
  let l = make_matrix (m+1) (n+1) (0,Diag) in
  for i = 1 to m do
    for j = 1 to n do
      l.(i).(j) <- maxi x y l i j
    done
  done;
(* et on parcourt le chemin a l'envers *)
  let longueur = (fst l.(m).(n)) and i = ref m and j = ref n in
  let result = make_vect longueur x.(0) and k = ref longueur in
  while (!k > 0) do
    if(snd l.(!i).(!j)) = Diag then
      ( decr i; decr j; decr k; result.(!k) <- x.(!i); )
    else if (snd l.(!i).(!j)) = Horiz then decr j
    else decr i
  done;
  result;;
let t1 = [|'a';'b';'c'|];;
let t2 = [|'a';'d';'b';'c'|];;
subseq_length t1 t2;;
subseq t1 t2;;
```

▶ **Question 5** La fonction prod calcule le produit de deux matrices entières.

```
t prod = fun a b ->
let na = vect_length a and pa = vect_length a.(0) in
let nb = vect_length b and pb = vect_length b.(0) in
if pa <> nb then failwith "matrices_incompatibles";
let c = make_matrix na pb 0 in
for i = 0 to na - 1 do
  for j = 0 to pb - 1 do
    for k = 0 to pa - 1 do
      c.(i).(j) <- c.(i).(j) + (a.(i).(k) * b.(k).(j))
    done
  done
done;
c;;
```

Le coût de cet algorithme naif est en $O(na * pb * pa)$ en temps et $O(na * pb)$ en espace.

▶ **Question 6** Considérons une suite de trois matrices de dimensions suivantes :

$$M_1 : 2 \times n$$
$$M_2 : n \times 2$$
$$M_3 : 2 \times n$$

Si on effectue le produit en utilisant le parenthésage $(M_1M_2)M_3$, on effectue $2 \times n \times 2$ multiplications puis $2 \times 2 \times n$, soit au total $8n$ multiplications d'entiers. Au contraire, si on considère le parenthésage $M_1(M_2M_3)$ alors $4n^2$ multiplications sont nécessaires. Le rapport entre les coûts deux alternatives est $n/2$.

▶ **Question 8**

```
let min_fun = fun f i1 i2 ->
    let i = ref i1 in let res = ref (f !i) in
    for j = i1 to i1 do
      if !res > (f j) then (res := f j; i := j; );
    done;
    !i;;
```

▶ **Question 9** Notre fonction utilise une sous-fonction p qui permet de calculer récursivement la valeur de $p_{i,j}$ pour $i \leq j$.

```
t optprod = fun d ->
let n = vect_length d - 1 in
  let rec p = fun i j ->
    let f = fun k ->
      (p i k ) + (p (k+1) j) + d.(i)*d.(k+1)*d.(j+1) in
    if i = j then 0
    else f (min_fun f i (j-1))
  in p 0 (n-1)
```

▶ **Question 10**

```
let optprod_dyn = fun d ->
    let n = vect_length d - 1 in
      let p = make_matrix n n 0 in
      for l = 1 to (n-1) do
        for i = 0 to (n-l-1) do
          let j = l + i in
          let f = fun k -> p.(i).(k)+p.(k+1).(j)+d.(i)*d.(k+1)*d.(j+1) in
          let k = min_fun f i (j-1) in
          p.(i).(j) <- f k
        done
      done;
      p.(0).(n-1);;
```

▶ **Question 11** La complexité est simple à calculer, vu que c'est trois boucles for imbriquées. La boucle intérieure (celle de min_fun consiste en $j - i = l$ opérations, répétés $n - l$ fois par la 2ème boucle. La seconde boucle est répétée $n - 1$ fois. La complexité totale est de $\frac{n(n^2-1)}{6}$ opérations (une opération consistant en 3 multiplications).

▶ **Question 12** Notre fonction optprod_tree se décompose en deux parties. La première est analogue au corps de la fonction optprod, à ceci près que, à chaque itération, elle stocke la valeur de k pour laquelle le minimum est trouvé dans un tableau auxilliaire s. La seconde partie est formée d'une fonction récursive auxilliaire qui construit l'arbre de parenthésage à partir de ce tableau.

```
let optprod_tree d =
  let n = vect_length d - 1 in
  let p = make_matrix n n 0 in
  let s = make_matrix n n (-1) in
  for l = 1 to n - 1 do
    for i = 0 to n - l - 1 do
      let j = l + i in
      let f k = p.(i).(k) + p.(k+1).(j) + d.(i) * d.(k+1) * d.(j+1) in
      let k = min_fun f i (j-1) in
      p.(i).(j) <- f k;
      s.(i).(j) <- k
    done
  done;

  let rec aux i j =
    if i = j then (Matrix i)
    else Product (aux i s.(i).(j), aux (s.(i).(j) + 1) j)
  in
  aux 0 (n-1)
;;
```