Licence informatique - L3 Année 2012/2013

Conception d'algorithmes et applications (LI325)

COURS 5

Résumé. Dans cette cinquième séance, nous continuons l'exploration des algorithmes de type Programmation Dynamique. Nous traiterons grâce à ce principe un problème numérique (multiplications de matrices enchaînées) et un problème issu de la théorie des mots (recherche d'une plus longue sous-séquence commune).

1. Programmation Dynamique appliquée à l'Algorithmique Numérique

1.1. **Multiplications Matricielles Enchaînées.** Nous allons ici illustrer le principe de la Programmation Dynamique par un algorithme qui résout le problème des multiplications matricielles enchaînées. On suppose que l'on dispose d'une séquence $(A_1, A_2, ..., A_n)$ de n matrices à coefficients entiers, et que l'on souhaite calculer le produit $A_1A_2...A_n$. Bien évidemment, il y a plusieurs façons de faire ce calcul qui dépendent de l'ordre dans lequel on fait les multiplications. Cet ordre est classiquement déterminé par un parenthésage qui indique les priorités. Par exemple, pour la séquence de matrices (A_1, A_2, A_3, A_4), le produit $A_1A_2A_3A_4$ peut être calculé de cinq façons distinctes : $(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, $((A_1(A_2A_3))A_4)$, $(((A_1A_2)A_3)A_4)$. On parlera alors de *produit de matrices entièrement parenthésé* (inductivement, un produit entièrement parenthésé est une matrice unique ou le produit entre parenthèses de deux produits matriciels entièrement parenthésés). La manière dont une suite de matrices est parenthésée fait varier le nombre d'opérations nécessaires pour obtenir le produit. Faisons l'hypothèse dans l'exemple suivant que le nombre de multiplications d'entiers effectuées pour multiplier deux matrices de dimensions respectives $p \times q$ (p lignes et q colonnes) et $q \times r$ est pqr (ce qui est le coût de l'algorithme naïf de multiplication de deux matrices).

Pour mettre en lumière les différents coûts induits par les différents parenthésages d'un produit de matrices, considérons une séquence (A_1, A_2, A_3) de trois matrices de dimensions respectives 10×100, 100×5 et 5×50. Si l'on fait le calcul suivant le parenthésage $((A_1A_2)A_3)$, on effectue $10.100.5 = 5000$ multiplications d'entiers pour calculer la matrice produit A_1A_2 de dimension 10×5, plus $10.5.50 = 2500$ multiplications scalaires pour multiplier cette matrice avec A_3, pour un total de 7 500 multiplications d'entiers. Si on multiplie selon le parenthésage $(A_1(A_2A_3))$, on effectue $100.5.50 = 25000$ multiplications d'entiers pour calculer la matrice produit A_2A_3 de dimension 100×50, plus $10.100.50 = 50000$ multiplications pour multiplier A_1 par cette matrice, pour un total de 75 000 multiplications scalaires. Le calcul du produit selon le premier parenthésage nécessite donc 10 fois moins de multiplications d'entiers.

Le problème des multiplications matricielles enchaînées peut être énoncé comme suit : étant donnée une séquence $(A_1, A_2, ..., A_n)$ de n matrices d'entiers où, pour $i = 1, 2, ..., n$, la matrice A_i est de dimension $p_{i-1} \times p_i$, comment parenthéser entièrement le produit $A_1A_2...A_n$ de façon à minimiser le nombre de multiplications scalaires.

Notez que, dans le problème des multiplications matricielles enchaînées, on ne multiplie pas les matrices. On cherche simplement un ordre de multiplication qui minimise le coût. Généralement, le temps passé à déterminer cet ordre optimal est plus que compensé par le gain de temps que l'on obtiendra quand on fera les multiplications matricielles proprement dites (par exemple, quand on fera 7 500 multiplications scalaires au lieu de 75 000).

1.1.1. *Sur le nombre de parenthésages différents.* Avant de résoudre le problème des multiplications matricielles enchaînées par le principe de la Programmation Dynamique, vérifions que passer en revue tous les parenthésages possibles ne donne pas un algorithme efficace. Soit $P(n)$ le nombre de parenthésages possibles d'une séquence de n matrices. Quand $n = 1$, il n'y a qu'une seule matrice et donc une seule façon de parenthéser entièrement le produit (remarquer que l'on devrait pour être cohérent écrire (A_i) au lieu de A_i). Quand $n > 1$, un produit matriciel entièrement parenthésé est le produit parenthésé de deux sous-produits matriciels entièrement parenthésés, respectivement de $(A_1, ..., A_k)$ et $(A_{k+1}, ..., A_n)$ où $k \in \{1, 2, ..., n-1\}$. On obtient donc la récurrence :

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

Le nombre $P(n+1)$ est appelé le *n-ième nombre de Catalan*, on peut montrer que $P(n) = \frac{1}{n}\binom{2(n-1)}{n-1}$ et que $P(n) = \Theta((1/n)^{(3/2)})4^n$. Donc le passage en revue de tous les parenthésages possibles produit un algorithme exponentiel.

1.1.2. *Structure d'un parenthésage optimal.* Nous notons $A_{i..j}$ avec $i \leq j$, la matrice $B = A_i A_{i+1} ... A_j$. On rappelle que, pour tout parenthésage du produit $A_i A_{i+1} ... A_j$, il existe une valeur k, telle que ce produit est le produit entre parenthèses de deux sous-produits matriciels entièrement parenthésés, respectivement de $(A_i, ..., A_k)$ et $(A_{k+1}, ..., A_j)$. Le coût de ce parenthésage est donc le coût du calcul de la matrice $A_{i..k}$, plus celui du calcul de $A_{(k+1)..j}$, plus celui de la multiplication de ces deux matrices.

Supposons qu'un parenthésage optimal de $A_i A_{i+1} ... A_j$ soit construit à partir de produits entièrement parenthésés de $(A_i, ..., A_k)$ et $(A_{k+1}, ..., A_j)$. Alors, le parenthésage induit de la sous-séquence «préfixe» $(A_i, A_{i+1}, ..., A_k)$ est forcément un parenthésage optimal de $(A_i, A_{i+1}, ..., A_k)$. En effet, s'il existait un parenthésage plus économique de $(A_i, A_{i+1}, ..., A_k)$, on pourrait substituer ce parenthésage dans le parenthésage optimal de $(A_i, A_{i+1}, ..., A_k)$ et on obtiendrait un autre parenthésage de $(A_i, ..., A_j)$ dont le coût serait inférieur à l'optimum, ce qui bien évidemment contradictoire. On peut faire la même observation pour le parenthésage de la séquence suffixe $(A_{k+1}, ..., A_j)$ induit par le parenthésage optimal de $(A_i, ..., A_j)$.

1.1.3. *Une solution récursive.* Considérons maintenant que le temps de calcul du produit de deux matrices A et B respectivement de dimensions $p \times q$ et $q \times r$ soit $f(p, q, r)$. Nous pouvons définir $m[i, j]$ le nombre minimal de multiplications scalaires nécessaires pour le calcul de la matrice $A_{i..j}$ récursivement de la manière suivante. Si $i = j$, la séquence est constituée d'une seule matrice $A_{i..j} = A_i$, et aucune multiplication n'est nécessaire pour calculer le produit. Donc, $m[i, i] = 0$ pour $i = 1, 2, ..., n$. Pour calculer $m[i, j]$ quand $i < j$, on exploite la propriété que l'on vient de remarquer dans la section précédente. Supposons que le parenthésage optimal sépare le produit en $(A_{i..k}).(A_{k+1..j})$, avec $i \leq k < j$. Alors, $m[i, j]$ est égal au coût minimal du calcul des sous-produits $A_{i..k}$ et $A_{k+1..j}$, plus le coût de la multiplication de ces deux matrices. Si l'on se rappelle que chaque matrice A_i est de dimension $p_{i-1} \times p_i$, on voit que le calcul de la matrice produit $(A_{i..k}).(A_{k+1..j})$ demande $f(p_{i-1}, p_k, p_j)$ multiplications scalaires. On obtient donc

$$m[i, j] = m[i, k] + m[k + 1, j] + f(p_{i-1}, p_k, p_j).$$

Cette équation récursive suppose que l'on connaisse la valeur de k, ce qui n'est pas le cas. Cela dit, il n'existe que $j - i$ valeurs possibles pour k, à savoir $k = i, i + 1, ..., j - 1$. Comme le parenthésage optimal doit utiliser une de ces valeurs pour k, il suffit de toutes les vérifier pour trouver la meilleure. Notre définition récursive du coût minimal de parenthésage du produit $A_{i..j}$ est donc :

$$m[i, j] = \begin{cases} 0 & \text{si } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j]\} + f(p_{i-1}, p_k, p_j) & \text{si } i < j \end{cases}$$

Les valeurs $m[i, j]$ donnent les coûts des solutions optimales des sous-problèmes. Mais cela ne nous permet pas de reconstruire un parenthésage optimal. Pour permettre la construction d'une

solution optimale, on a besoin de garder en mémoire pour chaque $i < j$ une valeur $s[i,j]$ telle que $m[i,j] = m[i,s[i,j]] + m[s[i,j]+1,j] + f(p_{i-1}, p_k, p_j)$.

1.1.4. *Calcul des coûts optimaux.* On va commencer par calculer le coût optimal par une approche itérative. Le pseudo-code suivant suppose que la matrice A_i est de dimension $p_{i-1} \times p_i$, pour $i = 1, 2, ..., n$. L'entrée est un tableau contenant les dimensions $p_0, p_1, ..., p_n$. La procédure utilise un tableau auxiliaire $m[1..n, 1..n]$ pour mémoriser les coûts $m[i,j]$ et un tableau auxiliaire $s[1..n, 1..n]$ qui mémorise pour chaque $i < j$ un indice k tel que $m[i,j] = m[i,k] + m[k+1,j] + f(p_{i-1}, p_k, p_j)$.

Algorithme 1 : ORDRE-CHAÎNE-MATRICES(P)

Entrées : Un tableau $P[0..n]$ contenant les dimensions $p_0, ..., p_n$.
Sorties : Rien mais on a rempli les tableaux m et s.

```
1  n := longueur[P] − 1
2  pour i allant de 1 à n faire
3      m[i, i] := 0
4  pour ℓ allant de 1 à n − 1 faire
5      pour i allant de 1 à n − ℓ faire
6          j := i + ℓ ; m[i, j] := ∞;
7          pour k allant de i à j − 1 faire
8              q := m[i, k] + m[k + 1, j] + f(P[i − 1], P[k], P[j]);
9              si q < m[i, j] alors
10                 m[i, j] := q;
11                 s[i, j] := k;
```

Analyse de l'algorithme ORDRE-CHAÎNE-MATRICES :
Preuve de Terminaison :
Evident, il n'y a que des boucles prédéfinies.
Preuve de Validité :
L'algorithme commence par l'affectation $m[i,i] := 0$, pour $i = 1, 2, ..., n$ (coûts minimaux pour les chaînes de longueur 1) aux lignes 2-3. Il utilise ensuite la récurrence pour calculer $m[i, i+1]$ pour $i = 1, 2, ..., n-1$ (coûts minimaux pour les chaînes de longueur $\ell = 1$) pendant la première exécution de la boucle des lignes 4-11. Au deuxième passage dans la boucle, il calcule $m[i, i+2]$ pour $i = 1, 2, ..., n-2$ (coûts minimaux pour les chaînes de longueur $\ell = 2$), et ainsi de suite. A chaque étape, le coût $m[i,j]$ calculé aux lignes 8-11 ne dépend que des éléments du tableau $m[i,k]$ et $m[k+1,j]$ déjà calculés. Comme nous n'avons défini $m[i,j]$ que pour $i \leq j$, seule la partie du tableau m strictement supérieure à la diagonale principale est utilisée.
Analyse de la Complexité en nombre d'additions :
On a clairement la valeur suivante pour le nombre d'additions : $\sum_{\ell=1}^{n-1} \sum_{i=1}^{n-\ell} \sum_{k=i}^{i+\ell-1} 2$. Un simple calcul montre que $\sum_{\ell=1}^{n-1} \sum_{i=1}^{n-\ell} \sum_{k=i}^{i+\ell-1} 2 = (n^3 - n)/3$. La procédure ORDRE-CHAÎNE-MATRICES fait donc $\Theta(n^3)$ additions. L'algorithme nécessite un espace de stockage $\Theta(n^2)$ pour les tableaux m et s. ORDRE-CHAÎNE-MATRICES est donc beaucoup plus efficace que la méthode en temps exponentiel consistant à énumérer tous les parenthésages possibles et à tester chacun d'eux.

1.1.5. *Construction d'une solution optimale.* Bien que ORDRE-CHAÎNE-MATRICES détermine le nombre optimal de multiplications scalaires nécessaires pour calculer le produit d'une suite de matrices, elle ne montre pas directement comment multiplier les matrices. Il n'est pas difficile de construire une solution optimale à partir des données calculées et mémorisées dans le tableau $s[1..n, 1..n]$. Chaque élément $s[i,j]$ contient la valeur k telle que le parenthésage optimal fractionne le produit en $(A_{i..k}).(A_{k+1..j})$. On sait donc que, pour le calcul optimal de $A_{1..n}$, la dernière multiplication matricielle sera le produit de $A_{1..s[1,n]}$ et $A_{(s[1,n]+1)..n}$. Les multiplications antérieures

peuvent être calculées récursivement ; en effet, $s[1, s[1, n]]$ détermine la dernière multiplication effectuée lors du calcul de $A_{1..s[1,n]}$ et $s[s[1, n] + 1, n]$ détermine la dernière multiplication effectuée lors du calcul de $A_{(s[1,n]+1)..n}$. La procédure récursive ci-après affiche un parenthésage optimal de $(A_i, A_{i+1}, ..., A_j)$, à partir du tableau s calculé par ORDRE-CHAÎNE-MATRICES et des indices i et j. L'appel initial AFFICHE-PARENTHÉSAGE-OPTIMAL$(s, 1, n)$ affiche un parenthésage optimal de $(A_1, ..., A_n)$.

Algorithme 2 : AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, i, j)

Entrées : Le tableau s et deux entiers.
Sorties : Rien mais on affiche un parenthésage optimal pour le calcul de $A_{i..j}$.

```
si i = j alors
    afficher("A"_i)
sinon
    afficher("(");
    AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, i, s[i, j]);
    AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, s[i, j] + 1, j);
    afficher(")");
```

Nous proposons maintenant une version récursive de l'algorithme. Elle est composée de deux parties, une partie initialisation (MÉMORISATION-CHAÎNE-MATRICES) et une partie construction récursive (RÉCUPÉRER-CHAÎNE).

Algorithme 3 : MÉMORISATION-CHAÎNE-MATRICES(P)

Entrées : Un tableau $P[0..n]$ contenant les dimensions $p_0, ..., p_n$.
Sorties : $m[i, j]$ le coût optimal du calcul $A_{i..j}$

```
n := longueur[P] − 1
pour i allant de 1 à n faire
    pour j allant de 1 à n faire
        m[i, j] := ∞
retourner RÉCUPÉRER-CHAÎNE(P, 1, n)
```

Algorithme 4 : RÉCUPÉRER-CHAÎNE(P, i, j)

Entrées : Un tableau $P[0..n]$ contenant les dimensions $p_0, ..., p_n$ et deux entiers i et j.
Sorties : $m[i, j]$ le coût optimal du calcul $A_{i..j}$

```
si m[i, j] < ∞ alors
    retourner m[i, j]
sinon
    si i = j alors
        m[i, j] := 0
    sinon
        pour k allant de i à j − 1 faire
            q := RÉCUPÉRER-CHAÎNE(P, i, k) +
            RÉCUPÉRER-CHAÎNE(P, k + 1, j) + f(P[i − 1], P[k], P[j])
            si q < m[i, j] alors
                m[i, j] := q ; s[i, j] := k
retourner m[i, j]
```

Analyse de l'algorithme :
Preuve de Terminaison :
Il suffit de remarquer que l'on appelle récursivement RÉCUPÉRER-CHAÎNE sur des instances où $j - i$ a diminué de 1 .
Preuve de Validité :
Immédiat par définition récursive de $m[i, j]$.
Analyse de la Complexité en nombre d'additions :
Identique à la version itérative.

2. Programmation Dynamique appliquée à l'Algorithmique sur les mots

2.1. **Plus Longue Sous-séquence Commune.** Le problème de la plus longue sous-séquence commune consiste, étant données deux séquences $X = (x_1, x_2, ..., x_m)$ et $Y = (y_1, y_2, ..., y_n)$, à trouver une sous-séquence (suite extraite) commune à X et Y de longueur maximale. Nous allons montrer que le problème de la plus longue sous-séquence commune, que l'on notera en abrégé PLSC, peut être résolu efficacement grâce à la Programmation Dynamique.

2.1.1. *Propriété d'une plus longue sous-séquence commune.* On pourrait essayer de résoudre le problème de la PLSC en énumérant toutes les sous-séquences de X en ordre décroissant suivant leur longueur et en les testant pour savoir si elles sont aussi des sous-séquences de Y, et en s'arrêtant à la première sous-séquence qui vérifie le test. Chaque sous-séquence de X correspond par sa fonction indicatrice à un sous-ensemble des indices $1, 2, ..., m$ de X. Il existe donc 2^m sous-séquences de X, de sorte que cette approche demande un traitement en temps exponentiel dans le pire des cas (c'est-à-dire si la PLSC est de longueur 1), ce qui rend cette technique inexploitable pour de longues séquences. Or, le problème de la PLSC possède une propriété qui va nous permettre de suivre un principe de Programmation Dynamique. Etant donnée une séquence $X = (x_1, x_2, ..., x_m)$, on définit le i-ème préfixe de X, pour $i = 0, 1, ..., m$, par $X_i = (x_1, x_2, ..., x_i)$. Par exemple, si $X = (A, B, C, B, D, A, B)$, alors $X_4 = (A, B, C, B)$ et X_0 représente la séquence vide.

Lemme 1. *Soient deux séquences $X = (x_1, x_2, ..., x_m)$ et $Y = (y_1, y_2, ..., y_n)$, et soit $Z = (z_1, z_2, ..., z_k)$ une PLSC de X et Y.*

(1) *Si $x_m = y_n$, alors $z_k = x_m = y_n$ et Z_{k-1} est une PLSC de X_{m-1} et Y_{n-1}.*
(2) *Si $x_m \neq y_n$ et $z_k \neq x_m$ alors Z est une PLSC de X_{m-1} et Y.*
(3) *Si $x_m \neq y_n$ et $z_k \neq y_n$ alors Z est une PLSC de X et Y_{n-1}.*

Preuve.

(1) Si $z_k \neq x_m$, on peut concaténer $x_m = y_n$ à Z pour obtenir une sous-séquence commune de X et Y de longueur $k + 1$, ce qui contredit l'hypothèse selon laquelle Z est une PLSC de X et Y. On a donc forcément $z_k = x_m = y_n$. Or, le préfixe Z_{k-1} est une sous-séquence commune de longueur $(k - 1)$ de X_{m-1} et Y_{n-1}. On va montrer que c'est une PLSC. Supposons, en raisonnant par l'absurde, qu'il existe une sous-séquence commune W de X_{m-1} et Y_{n-1} de longueur plus grande que $k - 1$. Alors, la concaténation de $x_m = y_n$ à W produit une sous-séquence commune à X et Y dont la longueur est plus grande que k, ce qui est contradictoire avec la maximalité de k.
(2) Si $z_k \neq x_m$, alors Z est une sous-séquence commune de X_{m-1} et Y. S'il existait une sous-séquence commune W de X_{m-1} et Y de taille supérieure à k, alors W serait aussi une sous-séquence commune de X et Y ce qui contredit l'hypothèse selon laquelle Z est une PLSC de X et Y.
(3) Identique mutatis mutandis au cas 2.

□

La caractérisation du lemme 1 montre qu'une PLSC de deux séquences contient une PLSC pour des préfixes respectifs X' et Y' des deux séquences X et Y. Nous dirons que le problème de la

PLSC possède la propriété de *sous-structure optimale*. C'est-à-dire que l'on peut construire une solution optimale à partir de sous-solutions optimales.

2.1.2. *Une solution récursive.* Grâce au lemme 1 nous pouvons construire récursivement une solution comme suit. Si $x_m = y_n$, on cherche une PLSC de X_{m-1} et Y_{n-1}. La concaténation de $x_m = y_n$ à cette PLSC engendre une PLSC de X et Y. Si $x_m \neq y_n$, alors on doit alors résoudre deux sous-problèmes : trouver une PLSC de X_{m-1} et Y, et trouver une PLSC de X et Y_{n-1}. La plus grande des deux PLSC est une PLSC de X et Y. Comme ces cas épuisent toutes les possibilités, on sait que l'une des solutions optimales des sous-problèmes doit servir dans une PLSC de X et Y. Appelons $c[i,j]$ la longueur d'une PLSC des séquences X_i et Y_j. Si $i = 0$ ou $j = 0$, l'une des séquences a une longueur nulle, et donc la PLSC est de longueur nulle. La sous-structure optimale du problème de la PLSC débouche sur la formule récursive :

$$c[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0, \\ c[i-1, j-1] + 1 & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i = y_j, \\ \max\{c[i, j-1], c[i-1, j]\} & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i \neq y_j. \end{cases}$$

On remarque que, dans cette définition récursive, que la condition $x_i = y_j$ restreint le nombre de sous-problèmes à considérer. En effet, on considère alors le sous-problème consistant à trouver la PLSC de X_{m-1} et Y_{n-1} et dans ce cas, on n'a donc pas besoin de trouver une PLSC de X_{m-1} et Y (ni de PLSC de X et Y_{n-1}). Il y a donc des sous-problèmes qu'il n'est pas nécessaire de traiter pour répondre au problème initial.

2.1.3. *Calcul de la longueur d'une PLSC.* En se rapportant à la définition récursive des $c[i,j]$, on peut faire appel au principe de la Programmation Dynamique pour calculer les solutions de manière itérative. La procédure LONGUEUR-PLSC prend deux séquences $X = (x_1, x_2, ..., x_m)$ et $Y = (y_1, y_2, ..., y_n)$ en entrées. Elle stocke les valeurs $c[i,j]$ dans un tableau $c[0..m, 0..n]$ dont les éléments sont calculés dans l'ordre des lignes. (Autrement dit, la première ligne de c est remplie de la gauche vers la droite, puis la deuxième ligne, etc.) Elle gère aussi un tableau $b[1..m, 1..n]$ pour simplifier la construction d'une solution optimale. Intuitivement, $b[i,j]$ pointe vers l'élément du tableau qui correspond à la solution optimale du sous-problème choisie pendant le calcul de $c[i,j]$. A la sortie de l'algorithme $c[m,n]$ contient la longueur d'une PLSC de X et Y.

Algorithme 5 : LONGUEUR-PLSC(X, Y)

Entrées : X et Y deux tableaux contenant respectivement $x_1, x_2, ..., x_m$ et $y_1, y_2, ..., y_n$
Sorties : Rien mais on a rempli les tableaux c et b.

```
1  pour i allant de 1 à m faire
2      c[i,0] := 0
3  pour j allant de 1 à n faire
4      c[0,j] := 0
5  pour i allant de 1 à m faire
6      pour j allant de 1 à n faire
7          si x[i] = y[j] alors
8              c[i,j] := c[i-1,j-1] + 1;
9              b[i,j] := " ↙ "
10         sinon
11             si c[i-1,j] ≥ c[i,j-1] alors
12                 c[i,j] := c[i-1,j];
13                 b[i,j] := " ↓ "
14             sinon
15                 c[i,j] := c[i,j-1];
16                 b[i,j] := " ← "
```

Analyse de l'algorithme LONGUEUR-PLSC :
Preuve de Terminaison :
Immédiat.
Preuve de Validité :
Elle découle naturellement de la définition inductive de $c[i, j]$.
Analyse de la Complexité en nombre de comparaisons entre un élément de X et un élément de Y : Le nombre de comparaisons faites est mn, puisque l'on fait une comparaison à chaque itération de la boucle interne ligne 7. On a donc un algorihtme en $\Theta(mn)$.

2.1.4. *Construction d'une PLSC.* Le tableau b construit par LONGUEUR-PLSC peut servir à retrouver rapidement une PLSC de $X = (x_1, x_2, ..., x_m)$ et $Y = (y_1, y_2, ..., y_n)$. On commence tout simplement en $b[m, n]$ et on se déplace dans le tableau en suivant les flèches. Chaque fois que nous rencontrons une $\swarrow$ dans l'élément $b[i, j]$, on sait que $x_i = y_j$ appartient à la PLSC. Cette méthode permet de retrouver les éléments de la PLSC dans l'ordre inverse. La procédure récursive suivante imprime une PLSC de X et Y dans le bon ordre. L'appel initial est AFFICHER-PLSC(b, X, longueur[X], longueur[Y]). La procédure prend un temps $O(m + n)$, puisqu'au moins l'un des deux indices i ou j est décrémenté à chaque étape de la récursivité.

Algorithme 6 : AFFICHER-PLSC(b, X, i, j)

Entrées : Le tableau b, le tableau X et deux entiers i, j
Sorties : Rien mais imprime une PLSC

```
1  si i = 0 ou j = 0 alors
2      Stop
3  si b[i, j] = " ↙ " alors
4      AFFICHER-PLSC(b, X, i − 1, j − 1);
5      afficher x[i];
6  sinon
7      si b[i, j] = " ↓ " alors
8          AFFICHER-PLSC(b, X, i − 1, j)
9      sinon
10         AFFICHER-PLSC(b, X, i, j − 1)
```