Fiche TD17 : Programmation dynamique

Exercice 1 *Partitions d'entiers*

Une partition d'un entier naturel n est une décomposition de n sous la forme d'une somme d'entiers strictement positifs. Deux partitions ne différant que par l'ordre des termes sont considérés comme étant égales. Par convention, la somme vide est l'unique partition de 0. On cherche à déterminer pour $n \in \mathbb{N}$ le nombre $p(n)$ de partitions de n.

1. Vérifier que $p(5) = 7$ en exhibant toutes les partitions de 5.

On note $s(n, k)$ le nombre de partitions de n ne faisant intervenir que des entiers inférieurs ou égaux à k.

2. Déterminer $s(0, k)$ et $s(n, 0)$ pour tous entiers n, k et montrer que pour tout $k \geq n$, $s(n, k) = p(n)$.
3. Exprimer $s(n, k)$ en fonction des $s(n - i, i)$ pour $i \in [\![1, k]\!]$.
4. En déduire un algorithme dynamique permettant de calculer $p(n)$ et déterminer sa complexité temporelle et spatiale. Peut-on améliorer cette dernière ?

Exercice 2 *Rendu de monnaie*

On considère un système de monnaie utilisant p types de pièces différentes de valeurs $S = (c_1, ..., c_p)$ rangées dans l'ordre croissant. Pour tout $n \in \mathbb{N}$, on note $f(n, S)$ le nombre minimal de pièces nécessaires pour décomposer n dans le système de monnaie S.

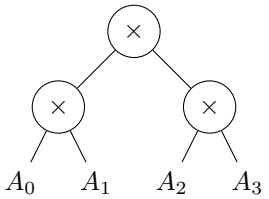
1. Etablir une relation de récurrence sur $f(n, S)$. *Indication : On pourra considérer* $S' = (c_1, ..., c_{p-1})$ *et exprimer* $f(n, S)$ *en fonction de* $f(n, S')$ *et* $f(n - c_p, S)$.
2. En s'aidant d'une matrice de taille $(n + 1) \times (p + 1)$ qui contient à la case (i, j) la valeur de $f(i, (c_1, ..., c_j))$, décrire un algorithme dynamique permettant de calculer $f(n, S)$.
3. Quelle est la complexité temporelle de cet algorithme ?
4. Quelle est sa complexité spatiale ? Peut on améliorer cette dernière ?
5. Expliquer comment modifier l'algorithme précédent pour calculer non seulement $f(n, S)$ mais encore une façon optimale de décomposer n selon le système de monnaie S (autrement dit, on souhaite savoir de combien de pièces de chaque type on a besoin pour aboutir à une somme égale à n dans une configuration où on utilise le moins de pièces possible).

Exercice 3 *Multiplication de matrices*

On considère le problème suivant : on cherche à calculer $\prod_{i=0}^{n-1} A_i$ où les A_i sont des matrices, pas forcément carrées. On considère que le produit de deux matrices de tailles respectives (p, q) et (q, r) nécessite pqr opérations. L'objectif est de trouver un parenthésage de ce produit qui soit optimal, c'est-à-dire qui minimise le nombre d'opérations à effectuer.

1. On considère 4 matrices A_0, A_1, A_2, A_3 de tailles respectives $(5, 2)$, $(2, 10)$, $(10, 4)$, $(4, 1)$. Indiquer le nombre d'opérations effectuées pour calculer le produit de ces matrices selon les parenthésages suivants :
   a) $(A_0A_1)(A_2A_3)$.
   b) $A_0(A_1(A_2A_3))$.
   c) $A_0((A_1A_2)A_3)$.
   d) $((A_0A_1)A_2)A_3$.
   e) $(A_0(A_1A_2)A_3)$.

Choisir un parenthésage revient à choisir un arbre binaire dont les noeuds internes sont étiquetés par $\times$ et les feuilles, lues de gauche à droite, sont $A_0, ..., A_{n-1}$. Par exemple, le parenthésage de 1.a) correspond à l'arbre :



2. On rappelle que le nombre de formes possibles pour un arbre binaire à k noeuds est $c_k = \frac{1}{k+1}\binom{2k}{k}$. Déterminer en fonction de n l'ordre de grandeur du nombre d'opérations qu'il faudrait effectuer pour déterminer un parenthésage optimal en utilisant la force brute.

Notons (l_i, c_i) les dimensions de A_i et notons $\text{opt}(i,j)$ le coût minimal du calcul de $\prod_{k=i}^{j} A_k$.

3. Expliquer pourquoi pour tout $i \leq j$ de $[\![0, n-1]\!]$ on a :

$$\text{opt}(i,j) = \min_{i \leq k < j} (\text{opt}(i,k) + \text{opt}(k+1,j) + l_i c_k c_j)$$

4. Décrire un algorithme utilisant une approche top-down permettant de calculer le nombre minimal d'opérations à effectuer pour calculer notre produit de matrices. Déterminer sa complexité.

5. Peut-on utiliser une approche bottom-up pour résoudre ce problème ? Si oui, expliquer comment.

6. Expliquer comment calculer un parenthésage optimal en plus du nombre minimal d'opérations nécessaires au calcul du produit.

Exercice 4 *Plus longue sous séquence croissante*

Une séquence est une suite finie d'entiers naturels deux à deux distincts $s = (s_0, ..., s_{n-1})$. La longueur d'une séquence est par définition son nombre d'éléments, n. Une sous-séquence de s est une sous suite de s. On cherche à déterminer la longueur maximale d'une sous-séquence croissante de s, notée $L(s)$ et à extraire de s une sous-séquence croissante maximale.

Par exemple, si $s = (7, 1, 2, 6, 4, 5, 8)$, $L(s) = 5$ et une sous séquence de s croissante maximale est $(1, 2, 4, 5, 8)$.

1. Minorer la complexité d'un algorithme brute-force (qu'on décrira sommairement) qui résout ce problème.

Pour tout $k \in [\![0, n-1]\!]$, on note $l(k)$ la longueur de la plus grande sous séquence croissante de s qui se termine en s_k.

2. Déterminer les $l(k)$ pour la suite $(10, 12, 2, 8, 3, 11, 7, 14, 9, 4)$.

3. Montrer que pour tout $0 \leq j < n-1$ on a : $l(k+1) = \begin{cases} 1 \text{ si } s_{k+1} < \min(\{s_0, ..., s_k\}) \\ 1 + \max(\{l(i) \mid 0 \leq i \leq k \text{ et } s_i \leq s_{k+1}\}) \text{ sinon.} \end{cases}$

4. En déduire un algorithme permettant de calculer le tableau des $l(k)$ pour $k \in [\![0, n-1]\!]$ et déterminer sa complexité en fonction de n.

5. En déduire un algorithme permettant de calculer $L(s)$ à partir de s et donner sa complexité.

6. Expliquer comment reconstruire une sous séquence de s croissante et de longueur maximale à partir du tableau des $l(k)$. Cette reconstruction déteriore-t-elle la complexité obtenue à la question précédente ?

Remarque : il est en fait possible d'obtenir un algorithme quasi linéaire en n pour ce problème. La recherche de plus longues sous séquences croissantes de séquences est liée au problème de la recherche de plus longues sous-séquences communes.

Exercice 5 *Plus longue sous séquence commune*

On se donne un alphabet Σ et deux mots $U = u_1 u_2 ... u_m$ et $V = v_1 ... v_n$ sur cet alphabet. On cherche à déterminer un plus long sous mot commun à U et V. Comme vu dans le DS5, un tel sous-mot existe et n'est pas forcément unique. On note U_i (respectivement V_i) le préfixe de U (respectivement V) de longueur i et pour tous $i, j \in [\![0, m]\!] \times [\![0, n]\!]$, on note $c(i,j)$ la longueur de la plus longue sous séquence commune à U_i et V_j.

1. Montrer que $c(i,j) = \begin{cases} 0 \text{ si } i = 0 \text{ ou } j = 0 \\ c(i-1, j-1) + 1 \text{ si } i, j > 0 \text{ et } u_i = v_j \\ \max(c(i, j-1), c(i-1, j)) \text{ sinon} \end{cases}$

2. En déduire un algorithme dynamique permettant de calculer la longueur d'un plus long sous mot commun à U et V. Déterminer sa complexité spatiale et temporelle.

3. Appliquer cet algorithme afin de déterminer la longueur d'un plus long sous mot commun à $BDCABA$ et $ABCBDAB$.

4. Expliquer comment modifier l'algorithme de la question 2 afin de calculer un plus long sous mot commun à U et V en sus de sa longueur. Cela déteriore-t-il la complexité de l'algorithme ?

5. Montrer qu'on peut améliorer la complexité spatiale de l'algorithme de la question 2 dans le cas où on ne cherche à calculer que la longueur d'un plus long sous mot commun à U et V.