TD - Introduction à la Programmation Dynamique

Enseignant : Areski Himeur
Sur la base des exercices de Marin Bougeret et Oliviez Bournez

DO5 - 2024

Par la suite, la notation $T[i..j]$ dénote le sous tableau compris entre les indices i et j inclus tous les deux et la notation $T[i..]$ dénote tous les éléments de T à partir de l'index i jusqu'à la fin du tableau.

Exercice 1. *Complexité d'un algorithme de programmation dynamique*
On considère la fonction suivante. Ne cherchez pas à comprendre ce qu'elle calcule, remarquez seulement qu'elle est correctement définie, au sens où elle termine toujours. Soit n_1 et n_2 deux entiers fixés et tels que $0 \leq n_1 \leq n_2$.

Algorithme 1 Une implémentation d'une fonction f

Entrée: Deux entiers i et j tels que $0 \leq i \leq n_1$; $0 \leq j \leq n_2$ et $0 \leq i \leq j$
  si $i = 0$ ou $i = j$ **alors**
    retourner 1
  sinon
    retourner $f(i, j-1) + f(i-1, j-1)$
  fin si

1. Transformer l'implémentation de f pour utiliser la programmation dynamique en ajoutant un tableau de mémoïsation des résultats.

2. Quelle est maintenant la complexité de f en fonction de n_1 et n_2 ?

Exercice 2. *Transformation d'un algorithme*
On considère la fonction f suivante. Ne cherchez pas à comprendre ce qu'elle calcule, remarquez seulement qu'elle est correctement définie, au sens où elle termine toujours. Soit n un entier fixé et // l'opérateur représentant la division entière.

Algorithme 2 Une implémentation d'une fonction inconnue f

Entrée: Un entier i tel que $0 \leq i \leq n$
  si $i = 0$ **alors**
    retourner 1
  sinon
    retourner $f(i//2) + f(i//3) + f(i-1)$
  fin si

1. Transformez l'implémentation pour utiliser la programmation dynamique.

2. Quel est le type de complexité de votre algorithme utilisant la programmation dynamique en fonction de n ?

Exercice 3. *Découpe de planche*
On considère une scierie qui connaît le prix de vente p_i pour une planche longueur i. Lorsqu'elle reçoit une planche de longueur n, elle peut soit en tirer le prix p_n, soit la découper en k morceaux de longueur $i_1, \ldots, i_k$ (avec $\sum_{\ell=1}^{k} i_l = n$) et en tirer $\sum_{\ell=1}^{k} p_{i_l}$. Définissons formellement ce problème de maximisation $\Pi_{DECOUPE}$:

> **Entrée :** un couple (p, i) tel que spécifié dans *decoupe*
> **Sortie :** un ensemble $S = \{i_1, \ldots, i_k\}$ d'entiers non nuls tels que $\sum_{x \in S} x = i$ ($S = \emptyset$ autorisé quand $i = 0$)
> **Objectif :** maximiser $f(S) = \sum_{x \in S} P[x]$

1. Écrire récursivement l'algorithme `decoupe(p, i)` qui étant donné un tableau p indexé de 1 à n tel que $p[i] = p_i$ et un i avec $0 \leq i \leq n$ calcule le meilleur prix que l'on puisse tirer d'une planche de taille i. Présentez l'arbre des appels récursifs pour $decoupe(p, 4)$ et estimez la complexité de l'algorithme.

N'utilisez pas pour l'instant la programmation dynamique. Par exemple, pour $p = [1, 5, 8, 9]$ et $n = 4$, $decoupe(p, 4)$ retourne 10.

2. Montrer que pour toute entrée (p, i) de $\Pi_{DECOUPE}$, $decoupe(p, i) = opt(p, i)$, c'est-à-dire que $decoupe(p, i)$ retourne bien une valeur optimale.

Aide : Étant donné une entrée (p, i) de découpe, considérez une solution optimale S^* et l^* un des entiers de S^*. Écrivons $S^* = \{l^*\} \cup S'$. Montrez que $f(S') \leq opt(p, i - l^*)$ puis que $opt(p, i) \leq \cdots$. En déduire que $decoupe(p, i) \geq opt(p, i)$.

3. Modifiez l'algorithme précédent pour utiliser la programmation dynamique. Nous appelons $DPdecoupe$ ce nouvel algorithme. Quelle est la complexité de $DPdecoupe(p, n)$?

4. Modifiez $DPdecoupe$ pour qu'il retourne également une solution optimale.

5. Proposez une version itérative de l'algorithme, toujours en utilisant la programmation dynamique.

Exercice 4. *Écriture d'un algorithme de programmation dynamique*

Soit n un entier positif. On considère le nombre de partitions de n, c'est-à-dire le nombre de façons d'écrire n comme une somme d'entiers positifs. Dans cet exercice, nous décidons de **tenir compte de l'ordre** ("1+2" n'est pas la même somme que "2 + 1"). Par exemple, il y a 8 partitions pour $n = 4$:

"4", "3 + 1", "2 + 2", "2 + 1 + 1", "1 + 3", "1 + 2 + 1", "1 + 1 + 2", "1 + 1 + 1 + 1"

On souhaite écrire un algorithme pour calculer le nombre de partitions de n.

Problème #Partition
Entrée : n un entier positif
Sortie : Le nombre de partitions de n (en tenant compte de l'ordre dans la somme)

1. Proposez un algorithme récursif pour résoudre le problème #Partition.

L'implémentation n'utilise pas pour le moment les principes de la programmation dynamique.

2. Modifiez l'algorithme précédent pour utiliser la programmation dynamique.

3. Quelle est la complexité de votre algorithme utilisant la programmation dynamique ?

4. Proposez une version itérative de l'algorithme, toujours en utilisant la programmation dynamique.

5. Proposez un algorithme similaire qui retourne également les solutions.

Par exemple, pour $n = 4$, les solutions seraient [″4″, ″3 + 1″, ″2 + 2″, ″2 + 1 + 1″, ″1 + 3″, ″1 + 2 + 1″, ″1 + 1 + 2″, ″1 + 1 + 1 + 1″].

Exercice 5. *Transformation en algorithme de programmation dynamique*

On considère le problème $P2||C_{max}$ d'ordonnancement de tâches indépendantes sur 2 machines. Les tâches sont ordonnancées depuis le temps 0, et sans interruptions entre les tâches :

Entrée : Un tableau t de n entiers positifs représentant les durées des tâches (la ième tâche dure $t[i]$)
Sortie : Une partition des tâches en deux ensembles M_1 et M_2 avec M_i représentant les indices des tâches sur la machine i et tel qu'en notant $C_i = \sum_{i \in M_i} t[i]$ la date de fin sur la machine i, $\max(C_1, C_2)$ soit minimum.

Algorithme 3 `P2CMAXAUX`

Entrée: Un tableau t de n entiers positifs représentant les durées des tâches, un entier positif $i \leq n$ et deux entiers positifs $load_1$ et $load_2$ inférieur à S

Sortie: La valeur optimale pour ordonnancer les taches de $t[i..n]$ en supposant que la machine i contienne déjà des tâches entre 0 et $load_i$

si $i = n$ **alors**
  retourner $max(load_1, load_2)$
sinon
  retourner $min(\texttt{P2CMAXAUX}(t, i+1, load1 + t[i], load2), \texttt{P2CMAXAUX}(t, i+1, load1, load2 + t[i]))$
fin si

Par exemple, pour $t = [10, 2, 2, 20, 5, 4]$, l'optimal est 22, correspondant à $M_1 = [2, 3]$ et $M_2 = [0, 1, 4, 5]$. Soit n la taille de t, et $S = \sum_{i=0}^{n-1} t[i]$ la somme des durées de toutes les tâches. Pour résoudre ce problème, on va brancher pour chaque i en essayant de mettre la tâche i sur M_1 ou sur M_2. Il faudra donc indiquer dans la récurrence la charge (appelée $load_i$) déjà accumulée sur chaque machine i. On obtient donc l'algorithme suivant :

1. Montrer comment résoudre optimalement une instance du problème $P2||C_{max}$ avec l'implémentation de `P2CMAXAUX`.

2. Transformez l'implémentation de la fonction `P2CMAXAUX` en une implémentation utilisant la programmation dynamique. Donnez la complexité de l'algorithme obtenue.

3. Transformez l'algorithme `P2CMAXAUX` (celle fournie de base, pas celle de la question précédente) en un algorithme `P2CMAXAUX-V2` qui calcul une solution et plus seulement sa valeur.

Exercice 6. *Décomposition d'une chaîne en blocs*

On considère un alphabet réduit deux caractères $A = \{a, b\}$. On considère le problème suivant de décomposition d'une chaîne en un minimum de patterns :

> **Entrée :** une chaine s sur l'alphabet A, et une liste de mots p (aussi sur l'alphabet A) appelés les *patterns*. On supposera que p contient (au moins) "a" et "b".
> **Sortie :** un découpage de s en k blocs $b_i, i \in [1..k]$ tels que $s = b_1.b_2 \cdots .b_k$, où . dénote la concaténation de chaînes et tel que pour tout $i \in k$, $b_i \in p$
> **Objectif :** minimiser k

Par exemple, pour $s =$ "baabaaaa" et $p =$ ["a","b","aab","aaaa","aba","aaa"], les deux décompositions suivantes sont possibles (mais il y en a d'autres)

- $s =$ "b"."aab"."aaaa" (on a donc $b_1 =$ "b", $b_2 =$ "aab", $b_3 =$ "aaaa", et donc une solution de coût 3, qui est d'ailleurs optimale)
- $s =$ "b"."a"."aba"."aaa" (on a donc $b_1 =$ "b", $b_2 =$ "a", $b_3 =$ "aba", $b_4 =$ "aaa", et donc une solution de coût 4)

On remarque que l'on a imposé que p contienne au moins "a" et "b" afin d'être sûr qu'il y a toujours une solution avec tous les b_i de taille 1.

1. Soit l'algorithme glouton G qui part de la gauche de s, et qui à chaque étape choisit un $b_i \in P$ de longueur la plus grande possible. Sur l'exemple ci-dessus $G(s, p)$ retournerait 3, puisqu'il construirait la première des deux solutions. Construire une instance (s', p') sur laquelle G est le plus mauvais possible.

Essayez d'abord d'obtenir une instance sur laquelle G n'est pas optimal, puis essayez de montrer que pour toute constance c il existe une instance (s', p') telle que $G(s', p') \geq c \times opt(s', p')$.

On va maintenant écrire un algorithme utilisant la programmation dynamique qui résout optimalement ce problème :

Dans l'exemple pour $s =$ "baabaaaa" et $p =$ ["a","b","aab","aaaa","aba","aaa"] ci-dessus, $minPattern(3)$ doit retourner 2.

Algorithme 4 minPattern

Entrée: Deux tableaux (s, p) entrés du problème et déclarés à l'extérieur et un entier positif $i \leq$ taille de s.

Sortie: Le plus petit nombre de blocs pour décomposer la sous chaine de s commençant en i. *Plus formellement, retourne le plus petit nombre k de blocs $b_1..b_k$ tels que $s[i..] = b_1..b_k$, et les b_i sont dans p.*

2. Écrire une implémentation de l'algorithme minPattern.

On ne demande pas d'ajouter le tableau de mémoisation, écrivez simplement un algorithme récursif qui peut facilement utiliser la programmation dynamique.

3. Donnez la complexité qu'aurait votre implémentation de minPattern si on la transformait en programmation dynamique.