Programmation dynamique

Exercice 1 Triangle de Pascal

On veut calculer les coefficients binomiaux $C_n^k = \left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}$. Rappellons les propriétés suivantes :

- $\left(\begin{array}{c} n \\ k \end{array} \right) = \left(\begin{array}{c} n-1 \\ k-1 \end{array} \right) + \left(\begin{array}{c} n-1 \\ k \end{array} \right)$ pour $0 < k < n$,
- $\left(\begin{array}{c} n \\ n \end{array} \right) = 1$ et $\left(\begin{array}{c} n \\ 0 \end{array} \right) = 1$.

Question 1.1 Donner un algorithme récursif du calcul de $\left(\begin{array}{c} n \\ k \end{array} \right)$. Evaluer sa complexité.

Correction

Fonction $bc(n, k)$

1. Si $k = 0$ ou $k = n$ alors retourner 1 ;
2. sinon retourner $bc(n-1, k-1) + bc(n-1, k)$.

□

Question 1.2 Ecrire l'algorithme qui retourne $\left(\begin{array}{c} n \\ k \end{array} \right)$ en utilisant la technique de la programmation dynamique. Evaluer sa complexité.

Correction

	0	1	2	3	⋯	$n-1$	n
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
⋮	⋮	⋮	⋮		⋱		
$n-1$	1	$n-1$	$\left(\begin{array}{c} n-1 \\ 2 \end{array} \right)$	$\left(\begin{array}{c} n-1 \\ 3 \end{array} \right)$	⋯	1	
n	1	n	$\left(\begin{array}{c} n \\ 2 \end{array} \right)$	$\left(\begin{array}{c} n \\ 3 \end{array} \right)$	⋯	n	1

Principe de l'algorithme

1. On rempli les k premières cases de chaque ligne de haut en bas.
2. On s'arrête à la ligne n
3. Temps : $T(n, k) = \mathcal{O}(nk)$.

Voici la procédure pour calculer $\begin{pmatrix} n \\ k \end{pmatrix}$

Soit $bc[n][n]$ le tableau qui stoque les coefficients.

1. pour i allant de 1 à n faire $bc[i][0] = 1$;
2. pour i allant de 1 à n faire $bc[i][i] = 1$;
3. pour i allant de 1 à n faire
   (a) pour j allant de 1 à j faire
   $$bc[i][j] = bc[i-1][j-1] + bc[i-1][j];$$
4. Retourner $bc[n][k]$

□

Exercice 2 Problème du stockage

Considérons n programmes $P_1, P_2, \ldots, P_n$ qui peuvent être stocker sur un disque dur de capacité D gigabytes.

— Chaque programme P_i a besoin s_i gigabytes pour être stocké et a une valeur v_i

— Tous les programmes ne peuvent pas être stockés sur le disque : $\sum_{i=1}^{n} s_i > D$.

Les programmes stockés dans le disque dur doivent maximiser la valeur totale, sans dépasser la capacité du disque dur. L'objectif est de concevoir un algorithme qui permet de calculer un tel ensemble.

Nous allons construire un tableau T dans lequel les lignes seront indexées par les programmes et les colonnes par les valeurs. L'élément $T[i, j]$ représentera la valeur maximale pour un disque dur de capacité j à l'aide des i premiers programmes.

Question 2.1 Donner la formule de récurrence.

Correction

Soit OPT une solution optimale avec un disque dur de capacité D et un ensemble $\mathcal{P}$ de n éléments. Notons $T[i, j]$ la valeur maximale pour un disque dur de capacité j à l'aide des i premiers objets.

1. Si le $n$ième élément appartient à la solution optimale, cela signifie, que la solution optimale privée du $n$ième élément est aussi une solution optimale avec un disque dur de capacité $D - s_n$ et un ensemble $\mathcal{P}$ privé du $n$ième élément.
   $$T[n, D] = v_n + T[n-1, D - s_n]$$
2. Si le $n$ième élément n'appartient pas à la solution optimale, cela signifie, que la solution optimale OPT est aussi une solution optimale avec un disque dur de capacité D et un ensemble $\mathcal{P}$ privé du $n$ième élément.
   $$T[n, D] = T[n-1, D]$$

Donc par récurence on peut en déduire pour tout $i \in \{1, \ldots, n\}$

Si $j < v_i$, alors $T[i, j] = T[i-1, j]$ sinon $T[i, j] = max(T[i-1, j], T[i-1, j - s_i] + v_i)$

On supposera que $\forall j \in \{1, \ldots, D\}$, on a $T[0, j] = 0$

□

Question 2.2 Donner l'algorithme utilisant la programmation dynamique.

Correction

Entrée : un ensemble d'objects $\mathcal{P} = \{1, \ldots, n\}$. L'objet i a une valeur v_i et un poids s_i.

Sortie : un entier

1. Intialiser tous les éléments du tableau T à zéro.
2. Pour tout i allant de 1 à n
   (a) Pour tout j allant de 1 à D

   2 (a).1 Si $j < v_i$, alors $T[i, j] = T[i-1, j]$
   sinon $T[i, j] = max(T[i-1, j], T[i-1, j-s_i] + v_i)$
3. Retourner $T[n, D]$

□

Question 2.3 Donner la complexité de cet algorithme.

Correction

La complexité de cet algorithme est $\mathcal{O}(nD)$ car

1. l'initialisation du tableau se fait en $\mathcal{O}(nD)$ opérations ;
2. L'instruction 2 (a).1 coûte $\mathcal{O}(1)$ opérations. Et elle est exécutée nD fois.

□

Remarque : Le problème de stockage peut se formuler sous forme du problème du Sac à Dos qui est un problème classique en informatique. Il modélise une situation analogue au remplissage d'un sac. Une personne veut remplir un sac à dos ne pouvant pas supporter plus d'un certain poids $D \in \mathbb{N}$, et elle dispose de n objets (On note l'ensemble des objects par $\mathcal{P} = \{1, \ldots, n\}$). Chaque objet i a une valeur $v_i \in \mathbb{N}\backslash\{0\}$ et un poids $s_i \in \mathbb{N}\backslash\{0\}$. Le problème est de trouver un ensemble d'objets tels que

- tous les objets de cet ensemble puissent être mis dans le sac.
- la somme des valeurs de ces objets soit maximale.

Exercice 3 Problème Le chemin le plus long dans un graphe

Soit $G = (V, E)$ un graphe orienté avec $V = \{v_1, \ldots, v_n\}$. On dit que G est ordonné si il vérifie les deux propriétés suivantes :

1. Chaque arc de ce graphe est de la forme $(i \to j)$ si $i < j$
2. Tous les sommets sauf le sommet v_n ont au moins un arc sortant.

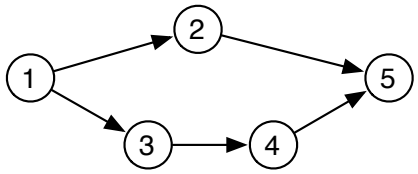
Ici, par souci de simplification, nous supposerons qu'il existe un chemin allant de v_i vers v_n pour tout $i = 1, \ldots, n$.

L'objectif est de trouver le chemin le plus long entre les sommets v_1 et v_n.

Question 3.1 Montrer que l'algorithme glouton suivant ne résoud pas correctement le problème.

1. $u \leftarrow v_1$;
2. $L \leftarrow 0$;
3. Tant qu'il existe un arc sortant du sommet u
   (a) choisir l'arc $(u \rightarrow v_j)$ tel que j est le plus petit possible
   (b) $u \leftarrow v_j$;
   (c) $L \leftarrow L + 1$;
4. retourner L

Correction



Le chemin retourné par l'algorithme est $\{1, 2, 5\}$. Le chemin le plus long est $\{1, 3, 4, 5\}$ □

Question 3.2 Donner la formule de récurrence qui permet de calculer la longueur du chemin le plus long commençant par v_1 finissant par v_ℓ.

Correction

Notons $long(v_\ell)$ la longueur du plus long chemin de v_1 vers v_ℓ.
- Si il n'existe pas de chemin de v_1 vers v_ℓ, alors par convention on supposera que $long(v_\ell) = -1$.
- Suppposons qu'il existe un chemin de v_1 vers v_ℓ.Notons $long(v_\ell)$ la longueur du plus long chemin de v_1 vers v_ℓ.
  - Si il n'existe pas de chemin de v_1 vers v_ℓ, alors par convention on supposera que $long(v_\ell) = -1$.
  - Suppposons qu'il existe un chemin de v_1 vers v_ℓ.
    Soit $L = \{v_1, \ldots, v_\ell\}$ le chemin le plus long de v_1 vers v_ℓ. Par définition du graphe orienté ordonné, les sommets de L sont dans $\{v_i : 1 \leq i \leq \ell\}$
    Notons v_j le sommet de L voisin de v_ℓ et L' le sous-chemin de v_1 vers v_j.
    Nous pouvons montrer que L' est le plus long chemin allant de v_1 finissant par v_j et que les sommets de L' sont dans $\{v_i : 1 \leq i \leq j\}$

$$long(v_\ell) = \begin{cases} 1 + max\{long(v_i) : i \in \Gamma^-(v_\ell)\} & si \quad \ell \neq 1 \\ 0 & si \quad \ell = 1 \end{cases}$$

□

Question 3.3 Donner un algorithme qui retourne la longueur du chemin le plus long commençant par v_1 finissant par v_n.

Correction

Soit $long : V \rightarrow \mathbb{N}$ un tableau d'entiers tel que $long(v_i)$ désigne la longueur du plus grand chemin de v_1 à v_i.
Entrée : un graphe orienté et ordonné G
Sortie : un entier

1. $long(v_1) \leftarrow 0$; *// O(1) opérations*
2. pour tout i allant de 2 à n faire $long(v_i) \leftarrow -1$; *// O(n) opérations*
3. pour tout i allant de 1 à n faire *// la boucle est exécutée O(n) fois*
   (a) pour tout $v_j \in \Gamma^+(v_i)$ faire *// la boucle est exécutée O(n) fois*
      i. si ($long(v_i) + 1 > long(v_j)$) alors $long(v_j) \leftarrow long(v_i) + 1$;
4. retourner $long(v_n)$;

Complexité : $O(n^2)$ opérations.

□

Question 3.4 Modifier l'algorithme précédent afin qu'il retourne le chemin.

Correction

Soit T un tableau où chaque élément $T[v_I]$ correspond un chemin de v_1 à v_i correspondant au chemin de plus grande la longueur.
Entrée : un graphe orienté et ordonné G
Sortie : un chemin

1. $long(v_1) \leftarrow 0$; $T[v_1] \leftarrow \{v_1\}$; *// O(1) opérations*
2. pour tout i allant de 2 à n faire *// la boucle est exécutée O(n) fois*
   (a) $long(v_i) \leftarrow -1$; *// O(1) opérations*
   (b) $T[v_i] \leftarrow \emptyset$; *// O(1) opérations*
3. pour tout i allant de 1 à n faire *// la boucle est exécutée O(n) fois*
   (a) pour tout $v_j \in \Gamma^+(v_i)$ faire *// la boucle est exécutée O(n) fois*
      i. si ($long(v_i) + 1 > long(v_j)$) alors
         $long(v_j) \leftarrow long(v_i) + 1$; *// O(1) opérations*
         $T[v_j] \leftarrow T[v_i] \cup \{v_j\}$; *// O(n) opérations au pire*
4. retourner $T[v_n]$;

Complexité : $O(n^3)$ opérations.

□

Nous allons considérer les graphes orientés et ordonnés possédant une fonction de poids w sur les arcs $w : E \to \mathbb{N}^+$.

L'objectif est de trouver le poids du chemin de poids maximal commençant par v_1 finissant par v_n (si il n'en existe pas, la valeur retournée doit être égale à $-\infty$). Formellement, on veut trouver $p_{max} = max\{\sum_{e \in \mathcal{P}} w(e) : \mathcal{P}$ est un chemin de v_1 à $v_n\}$

Question 3.5 Donner la formule de récurrence permettant de calculer le chemin de poids maximum commençant par v_1 finissant par v_n.

Correction

Notons $poids(v_\ell)$ la longueur du plus long chemin de v_1 vers v_ℓ.

Si il n'existe pas de chemin de v_1 vers v_ℓ, alors par convention on supposera que $poids(v_\ell) = -\infty$.

Suppposons qu'il existe un chemin de v_1 vers v_ℓ. Soit $L = \{v_1, \ldots, v_\ell\}$ le chemin de poids maximum de v_1 vers v_ℓ. Par définition, par la définition du graphe orienté ordonné,

les sommets de L sont dans $\{v_i : 1 \leq i \leq \ell\}$

$$poids(v_\ell) = \begin{cases} max\{poids(v_i) + w(v_i, v_\ell) : v_i \in \Gamma^-(v_\ell)\} & si \quad \ell \neq 1 \\ 0 & si \quad \ell = 1 \end{cases}$$

□

Question 3.6 En déduire l'algorithme.

Correction

Soit $long : V \to \mathbb{N}$ un tableau d'entiers tel que $long(v_i)$ désigne la longueur du plus grand chemin de v_1 à v_i.
Entrée : un graphe orienté et ordonné G
Sortie : un entier

1. $long(v_1) \leftarrow 0$; *// O(1) opérations*
2. pour tout i allant de 2 à n faire $long(v_i) \leftarrow -1$; *// O(n) opérations*
3. pour tout i allant de 1 à n faire *// la boucle est exécutée O(n) fois*
   (a) pour tout $v_j \in \Gamma^+(v_i)$ faire *// la boucle est exécutée O(n) fois*
      i. si ($poids(v_i)+w(v_i, v_j) > poids(v_j)$) alors $poids(v_j) \leftarrow poids(v_i)+w(v_i, v_j)$;
4. retourner $poids(v_n)$;

Complexité : $O(n^2)$ opérations.

□

Maintenant, l'objectif est de trouver le poids du chemin de poids maximal commençant par v_1 composé de k arcs. Nous utiliserons un tableau T dont la valeur de l'élément $T[i, \ell]$ correspond au poids du chemin de poids maximal commençant par v_1 finissant par v_i et composé de ℓ arcs.

Question 3.7 Donner la formule de récurrence qui permet de calculer le chemin de poids maximal de ℓ arcs commençant par v_1 finissant par v_i . En déduire l'algorithme.

Exercice 4 Planning

Considérons un chef de projet qui doit gérer une équipe en lui affectant un projet qui dure une semaine. Le chef de projet doit choisir si il prend un projet « stressant » ou « non-stressant » pour la semaine.

1. Si le chef de projet choisit le projet qui n'est pas « stressant » durant la semaine i, alors l'entreprise reçoit un revenu ℓ_i.
2. Si le chef de projet choisit le projet qui est « stressant » durant la semaine i, alors l'entreprise reçoit un revenu h_i et l'équipe ne travaille pas durant la semaine $i - 1$

Exemple : Si chef de projet choisit de se reposer à la semaine 1, puis de prendre le projet « stressant » à la semaine 2, puis de prendre les projets « non-stressant » à la semaine 3 et 4. Le revenu total est de 70 et il correspond au maximun.

	semaine 1	semaine 2	semaine 3	semaine 4
ℓ	10	1	10	10
h	5	50	5	1

Question 4.1 Montrer que l'algorithme suivant ne résoud pas correctement le problème.

1. pour chaque itération $i = 1, \ldots, n$
   (a) si $h_{i+1} > \ell_i + \ell_{i+1}$ alors
      i. Choisir « Ne pas travailler à la semaine i »
      ii. Choisir « le projet « stressant » à la semaine i+1 »
      iii. Continuer à l'itération $i + 2$
   (b) sinon
      i. Choisir « le projet « non-stressant » à la semaine i »
      ii. Continuer à l'itération $i + 1$

Question 4.2 Donner un algorithme qui retourne le revenu maximal que peut obtenir le chef de projet.

Exercice 5 Multiplications chaînées de matrices.

On veut calculer le produit de matrices $M = M_1 M_2 \cdots M_n$. Multiplier une matrice $p \times q$, par une matrice $q \times r$ en utilisant la méthode standard nécessite pqr produit scalaire.

Question 5.1 Considérons 4 matrices $A : 20 \times 5$, $B : 5 \times 100$, $C : 100 \times 8$, $D : 5 \times 30$. On veut calculer le produit $ABCD$. En fonction des parenthétisations, le nombre de produits varie.

Déterminer le nombre de produits pour calculer $ABCD$, si on utilise les parenthétisations suivantes : $((AB)C)D$ ou $(A(BC))D$

Correction

1. le produit de matrices utilisant cet ordre $((AB)C)D$ nécessite 30800 produits scalaires, c'est-à-dire,
   — $AB : 20 \times 5 \times 100 = 10000$
   — $((AB)C) : 20 \times 100 \times 8 = 16000$
   — $((AB)C)D : 20 \times 8 \times 30 = 4800$
2. le produit de matrices utilisant cet ordre $(A(BC))D$ nécessite 9600 produits scalaires, c'est-à-dire,
   — $BC : 5 \times 100 \times 8 = 4000$
   — $((A(BC)) : 20 \times 5 \times 8 = 800$
   — $((AB)C)D : 20 \times 8 \times 30 = 4800$

□

L'objectif est de concevoir un algorithme de meilleure parenthétisation qui permet de minimiser le nombre de produits scalaires.

Nous noterons que la matrice M_i est de dimension $d_{i-1} \times d_i$.

Définissons le nombre minimal de produits scalaires nécessaires pour évaluer le produit des matrices $M_i \times M_{i+1} \ldots M_{j-1} \times M_j$ par $c(i, j)$.

Question 5.2 Ecrire une formule de récurrence pour calculer $c(i, j)$.

Correction

Supposons que la meilleure façon de parenthéser $M_i \ldots M_j$ soit $(M_i..M_k)(M_{k+1} \ldots M_j)$. La matrice $M_i \ldots M_k$ est une matrice $d_{i-1} \times d_k$ et $M_{k+1} \ldots M_j$ est une matrice $d_k \times d_j$. Le

produit de ces deux matrices nécessite $d_{i-1}d_kd_j$ produits scalaires. Au total, le nombre total de produits scalaires pour calculer $M_iM_{i+1}\dots M_{j-1}M_j$ est $c(i,k)+c(k+1,j)+d_{i-1}d_kd_j$.

On obtient

$$c(i,j)=\begin{cases}\min_{i\le k<j}(c(i,k)+c(k+1,j)+d_{i-1}d_kd_j) & \text{si } i<j.\\ 0 & \text{si } i=j.\end{cases}$$

□

Question 5.3 Ecrire un algorithme utilisant la programmation dynamique

Correction

Donnée : une suite de matrices $M_1\dots M_n$ avec la matrice M_i de dimention $d_{i-1}\times d_i$.

1. initialiser tous les éléments de la matrice à $\infty+$
2. pour i à 1 allant n faire $c(i,i)=0$;
3. pour ℓ à 1 allant n faire
   (a) pour i à 1 allant $n-\ell$ faire
      i. pour k à 1 allant ℓ faire
         $c(i,i+\ell)=\min(c(i,i+\ell),c(i,i+k)+c(i+k+1,i+\ell)+d_{i-1}d_{i+k}d_{i+\ell})$;

□