

Terminale NSI

Programmation Dynamique - Travaux dirigés

qkzk

2020/04/29

Introduction

Dans ce T.D. nous allons résoudre quelques problèmes classiques en utilisant la **programmation dynamique**.

1. Les pyramides de nombres

Une pyramide de nombre est un graphe donc les sommets sont des nombres.

Chaque sommet d'un même niveau a deux arrêtes vers le bas.

Deux sommets voisins d'un même niveau sont reliés à un même sommet du niveau suivant.

```
0      3
1     2 4
2    2 7 5
3   9 5 4 6
```

Ici on peut suivre le chemin 3 - 4 - 7 - 5 mais pas 3 - 2 - 5 : les sommets 2 et 5 ne sont pas reliés.

Objectif :

Déterminer la valeur maximale de la somme des chemins traversant une pyramide de nombre.

Dans le premier exemple, le chemin 1 - 8 - 2 - 5 à pour somme 16, qui est maximale.

Algorithme naïf :

Il consiste à examiner tous les chemins possibles, et choisir celui qui a le plus grand total. En général, quand la pyramide a n niveaux, il y a 2^{n-1} chemins et $2^n - 2$ calculs à effectuer.

Donc l'algorithme naïf est en temps exponentiel en n .

Résoudre les pyramides de nombres avec un algorithme récursif

1. Déterminer le chemin optimal de la seconde pyramide de nombres.
2. Proposer un algorithme récursif.

On notera :

x la position, $v(x)$ la valeur, $c(x)$ la somme maximale quand on descend à partir de x .

3. Est-ce qu'on effectue plusieurs fois le même calcul ? Est-ce optimal ?

En utilisant la progression dynamique.

4. Rappeler les deux principes à respecter avant d'envisager la progression dynamique. Sont-ils respectés ici ?

Nous allons construire un algorithme différent. Il ne part plus de la pointe mais de la base.

Pour cela, on calcule, pour chaque étage *depuis la base* la somme maximale qu'on peut atteindre *depuis ce noeud*.

Par exemple,

- partant du 6 à la dernière ligne : on peut atteindre 6.
- partant du 7 à la 3^{ème} ligne on peut atteindre 7 + 5 et 7 + 4. La valeur maximale est 7+5+12. On inscrit 12 à cette ligne.

5. Compléter la pyramide des valeurs “sommés montantes” (il faut bien leur donner un nom !) du second exemple.

Sommés montantes

0		-	
1	-	-	
2	-	12	-
3	9	5	4 6

6. Proposer un algorithme calculant itérativement ces sommés montantes. Où la somme maximale est-elle référencée ?
7. Combien de sommés sont-elles effectuées ?
8. Vérifier sur les exemples.

On envisage d’implémenter cet algorithme (lire : vous le ferez en TP).

8. Choisir une structure de donnée adaptée pour les pyramides à cet algorithme.
9. **Complément pas indispensable.** On obtient bien la somme maximale, mais pas le chemin qui y mène. Que pourrait-on ajouter pour l’obtenir ?

Remarque finale : les pyramides de nombres sont des graphes. Alors ce qu’on vient d’effectuer est un parcours. Il visite tous les sommés et toutes les arrêtes mais évite d’emprunter certains chemins. La version récursive initiale visitait tous les chemins, d’où sa lenteur.

3. Le rendu de monnaie : combien de manières différentes de rendre la monnaie

Bien que le thème soit similaire, ce problème est différent de celui abordé en première.

Problème : Étant donné un jeu de pièces S combien existe-t-il de manière de rendre la somme n ?

On note $C(S, n)$ le nombre de sommés qu’on peut former.

- Par exemple si $S = [1, 2, 5]$ et $n = 5$.
 $n = 1 + 1 + 1 + 1 + 1 = 1 + 1 + 1 + 2 = 1 + 2 + 2 = 5$. Il existe 4 manières différentes.

$$C([1, 2, 5], 5) = 4$$

- Si $n = 0$ alors il existe une manière : on ne rend rien.
- Si $n = 1$ alors il existe une manière : on rend une pièce.

En quoi est-ce un problème de programmation dynamique ?

Exhibons les sous-problèmes :

Pour une pièce p du jeu S , si $p \leq n$ alors :

- Je peux décider prendre p , alors j’ai $S - p$ à rendre encore.
- Si je ne la prends pas, j’ai autant de solutions possibles que si j’avais retiré p de S .

$$C(S, n) = C(S, n - p) + C(S \setminus \{p\}, n)$$

On constate que :

- les sous-problèmes ne sont pas indépendants,
- les sous structures (rendre $n - p$ avec S et rendre n avec $S \setminus \{p\}$) sont optimales.

Donc : c’est bien un problème de programmation dynamique et nous avons déjà la relation de récurrence.

Afin d’illustrer nous allons remplir un tableau pour les données du premier exemple.

Le tableau ci-dessous se remplit par ligne.

- La première ligne indique le montant n à atteindre.

- La première colonne indique le jeu de pièces S dont on dispose
- Les cases intérieures contiennent les valeurs de $C(S, n)$.
- Par exemple, si je n'ai aucune pièce, $S = []$.
 - Avec 0 à rendre, j'ai 1 manière de rendre la monnaie,
 - Avec $n = 1, 2, 3, \dots$ je n'ai aucune manière de rendre la monnaie.
- Avec $S = [1, 2]$.
 - Si $n = 0$ il y a une manière.
 - Si $n = 1$. Nous allons utiliser la relation de récurrence :
 Si je prends la pièce 1, il me reste 0 à rendre. Je lis la valeur $C = 1$ dans le tableau :
 Si je ne prends pas la pièce 1, il me reste 1 à rendre avec $S = []$. Je lis la valeur $C = 0$ dans le tableau.
 J'additionne ces deux nombres et j'écris la valeur **1** dans le tableau.

$S \setminus n$	0	1	2	3	4	5
$[\]$	1	0	0			
$[1]$	1	1				
$[1, 2]$						
$[1, 2, 5]$						

1. Compléter le tableau
2. Proposer un algorithme pour résoudre le problème.

2. Le rendu de monnaie : nombre minimal de pièces

On a déjà abordé ce problème longuement en première. Il figure aussi au programme de terminale mais avec un contexte très différent.

Attention, il est nettement plus difficile que le précédent.

Rendu de monnaie :

On dispose d'un jeu de pièces, par exemple $[1, 2, 5, 10]$ et d'une somme à construire, par exemple $S = 12$.

Quel est le nombre minimal de pièces nécessaires pour atteindre S ?

En première on a résolu ce problème avec un algorithme glouton qui fonctionnait parfaitement pour un système de pièces **canonique**.

Système de pièces canonique :

La somme des n premières pièces (par ordre croissant) est inférieure à la pièce $n + 1$

Que se passe-t-il si le système n'est plus canonique ?

Par exemple, système monétaire anglais pré 1971 qui adoptaient les multiples suivant du *penny* : 1, 3, 6, 12, 24, 30. Avec ce système, l'algorithme glouton décompose $48p$ en $30p + 12p + 6p$ alors que la décomposition optimale est $24p + 24p$.

Nous allons proposer une démarche moins rapide que l'algorithme glouton mais qui retourne toujours la solution optimale.

1. Quel algorithme naïf pourrait-on envisager ? Inconvénient ?
2. Peut-on appliquer ici la programmation dynamique ?
3. On note S la somme, n la somme, S le jeu de pièces dont on dispose et $f(n, S)$ le nombre minimal de pièces à utiliser.
 Proposer une relation de récurrence sur f .
4. Proposer un algorithme récursif pour résoudre le problème du rendu de monnaie.

3. Calcul des combinaisons

Les *combinaisons* sont des entiers déjà rencontrés en mathématiques qui apparaissent dans beaucoup de problèmes :

- développer $(a + b)^n$ grâce à la formule du *binôme de Newton*,
- calculer une probabilité avec une loi binomiale,

Définition :

Le nombre de combinaisons de k parmi n est le nombre de mots de n qu'on peut écrire avec seulement deux symboles et en utilisant k fois le premier

Exemple : Avec les symboles 0 et 1, combien de mots de quatre lettres peut-on former qui comportent deux 0 ?

Les mots sont : 0011, 0101, 0110, 1001, 1010 et 1100. Aussi $\binom{4}{2} = 6$.

Pour être certain de n'en oublier aucun il suffit de les donner par ordre croissant.

Propriétés élémentaires :

- On a toujours une possibilité de choisir tous les objets ou de n'en choisir aucun donc $\binom{n}{0} = \binom{n}{n} = 1$
- Si l'on choisit 1 objet ou si on en délaisse un, il y a n tirages possibles donc $\binom{n}{1} = \binom{n}{n-1} = n$

Formule générale :

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Cette formule est inutilisable en informatique.

Certes tous les nombres sont entiers, mais ils deviennent vite énormes.

Les combinaisons vérifient la **Formule du triangle de Pascal**

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

La formule précédente permet, de proche en proche, de construire le **Triangle de Pascal**

Dans le triangle ci-dessous, cela signifie :

- qu'on remplit les lignes une par une,
- qu'on ajoute deux valeurs voisine d'une même ligne pour obtenir celle sous la valeur de droite.

Par exemple le 3 est obtenu en faisant $1 + 2 = 3$ (ses voisins du dessus)

0. Compléter le triangle de Pascal suivant

n\k	0	1	2	3	4	5	6	7
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4								
5								
6								
7								

Problème : construire un algorithme efficace pour calculer les combinaisons.

On utilise le triangle de Pascal pour résoudre le problème

Voici les propriétés que nous allons utiliser :

- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ pour $0 < k < n$,

- $\binom{n}{0} = \binom{n}{n} = 1$
1. Donner une fonction récursive (en langage naturel ou en Python) qui calcule $\binom{n}{k}$
 2. Construire l'arbre des appels récursifs du calcul de $\binom{5}{3}$
(prévoir de la place, le nombre total d'appels récursifs est $2\binom{n}{k} - 2$).
 3. proposer un algorithme utilisant la programmation dynamique qui calcule $\binom{n}{k}$ en utilisant le triangle de Pascal.
 4. Est-il nécessaire de mémoriser *tout* le tableau ?
 5. Doit-on remplir entièrement chaque ligne ?
-

4. Découpe de planches

Lorsqu'elle reçoit en entrée une planche de longueur n , elle peut

- soit en tirer directement le profit/prix p_n ,
- soit chercher à la découper en k morceaux pour en tirer plusieurs (sous) planches de longueur i_1, i_2, \dots, i_k (avec $i_1 + i_2 + \dots + i_k = n$) et obtenir comme profit la somme $p_{i_1} + p_{i_2} + \dots + p_{i_k}$ des prix de vente des sous-planches.
- Le problème de la scierie est alors de déterminer la solution qui lui garantit un profit maximal.

Exemple

longueur i	1	2	3	4	5	6	7	8	9	10
prix p_i	1	5	8	9	10	17	17	20	24	30

1. énumérer tous les prix possibles pour une planche de longueur 4.
2. Quel est la solution optimale ?

On note r_n le profit maximal que l'on peut atteindre pour une planche de longueur n .

3. Proposer une relation de récurrence entre r_n et les valeurs inférieures.
4. Proposer une fonction récursive `coupe` qui calcule r_n en fonction des paramètres p (le tableau des prix) et n (la longueur de la planche).
5. Construire l'arbre des appels récursifs de `coupe(p, 4)`
6. Modifier votre fonction précédente pour qu'elle utilise une solution mémorisée.
 1. de haut en bas
 2. de bas en haut