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.

   Correction

   `3 + 4 + 7 + 5 = 19`

   Correction

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.

   Correction

   définition récursive de $c(x)$:

   $c(x) = v(x)$ pour toute position x situé au rez-de chaussée de la pyramide

   $c(x) = v(x) + \max(c(g(x)), c(d(x)))$ pour toute autre position x, où $g(x)$ et $d(x)$ sont les positions inférieures gauche et droite sous la position x.

   Correction

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

   Correction

   Cet algorithme effectue toujours beaucoup trop de calculs.

   La valeur 7 de la troisième ligne est visitée une fois en venant du 2 et une fois en venant du 4 (de la première ligne).

   Correction

En utilisant la progression dynamique.

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

   Correction

   - *Chevauchement des sous-problèmes*

     C'est bien le cas, plusieurs chemins passent au même endroit. Par exemple `3 - 4 - 7 - 4` et `3 - 2 - 7 - 4`.

   - *Sous structure optimale*

     Une fois qu'on a la somme maximale qu'on peut atteindre pour un noeud, il est inutile de la recalculer pour un autre

   Correction

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$^{\text{ième}}$ 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 "sommes montantes" (il faut bien leur donner un nom !) du second exemple.

```
Sommes montantes
0       _
1     _   _
2    _  12  _
3   9  5  4  6
```

   Correction

```
Sommes montantes ex 2
0       19
1     14  16
2   11  12  11
3  9   5   4   6
```

   Correction

6. Proposer un algorithme calculant itérativement ces sommes montantes. Où la somme maximale est-elle référencée ?

   Correction

   On crée une pyramide de mêmes dimensions remplie de zéros. L'étage le plus bas est initialisé avec les valeurs de la pyramide

   pour chaque étage à partir de l'avant dernier, jusqu'au premier

   - La valeur de la somme montante à une position est la valeur de la pyramide + le max des fils gauche et droit de la somme montante les fils gauche et droit sont : 1 étage plus bas,

   La somme maximale est référencée tout en haut de la pyramide **Correction**

7. Combien de sommes sont-elles effectuées ?

   Correction

Une somme par élément de la pyramide sauf la dernière ligne.

Donc $n(n-1)/2$ sommes et $n(n-1)/2$ maximums.

Correction

8. Vérifier sur les exemples.

   Correction

   Voir plus haut

   Correction

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.

   Correction

   On peut utiliser un dictionnaire ou une double liste. Chaque étage dans une liste.

   Correction

9. **Complément pas indispensable**. On obtient bien la somme maximale, mais pas le chemin qui y mène. Que pourrait-on ajouter pour l'obtenir ?

   Correction

   On peut conserver l'information du fils qui a fourni la somme la meilleure. Par exemple sa valeur ou sa position.

   Correction

Remarque finale : les pyramides de nombres sont des graphes. Alors ce qu'on vient d'effectuer est un parcours. Il visite tous les sommets 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 sommes 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 possibiles 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 $n = 1, 2, 3, \ldots$ 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 \ n	0	1	2	3	4	5
[ ]	1	0	0			
[**1**]	1	**1**				
[1, **2**]						
[1, 2, **5**]						

1. Compléter le tableau

   correction

	0	1	2	3	4	5
[ ]	1	0	0	0	0	0
[**1**]	1	1	1	1	1	1
[1, **2**]	1	1	2	2	3	3
[1, 2, **5**]	1	1	2	2	3	4

   correction

2. Proposer un algorithme pour résoudre le problème.

   correction

```
pour toute pièce p de S,
    pour tout reste à rendre r entre 1 et n,
        C(S, r) = C(S - {p}, r) + C(S, r - p)
```

   correction

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 ?

   Correction

   On pourrait calculer toutes les sommes qui valent la somme à rendre Ensuite on choisit parmi ces dernières celle qui est optimale.

   Dès qu'on a beaucoup de pièces, l'algorithme explose. Le nombre de sommes à envisager croît très rapidement.

   Autre problème, on calcule plusieurs fois la même somme.

   Correction

2. Peut-on appliquer ici la programmation dynamique ?

   Correction

   - Sous structure optimale :

     Si le nombre minimal pièces pour rendre un montant S est n et que la pièce de valeur la plus élevée vaut x alors le nombre minimal de pièces pour rendre $S - x$ est $n - 1$.

     Si ça n'était pas le cas on pourrait trouver un moyen de rendre $S - x$ avec moins de $n - 1$ pièces.

     On ajoute alors une pièce de valeur x et on obtient un montant $S - x + x = S$ avec moins de n pièces. C'est une contradiction.

   - _Chevauchement des sous problèmes.

     Lorsqu'on cherche un nombre minimal de pièces à rendre S avec les pièces, on peut décider de prendre la pièce c ou de ne pas la prendre.

     Ces deux problèmes ne sont pas indépendants.

   Correction

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.

   Correction

   On considère une configuration optimale permettant de rendre n et une pièce de valeur c.

   Si $c > n$, c ne figure pas dans la configuration.

   Sinon, deux cas se présentent : c figure dans la configuration optimale ou c n'y figure pas.

   on a :

   $$f(n, S) = \min\left(1 + f(n - c, S), f(n, S \setminus \{c\})\right)$$

   Correction

4. Proposer un algorithme récursif pour résoudre le problème du rendu de monnaie.

   Correction

   Système : le jeu de pièces, n le montant, p le numéro d'une pièce dans le système.

   On référence les états possibles dans une matrice de taille : `(nb de pieces + 1) * (montant + 1)`

```
Rendu de monnaie (systeme de pieces, montant)

pour chaque piece d'indice p du jeu,
  pour chaque reste r entre 1 et le montant,
    trois cas :
    * la piece vaut le reste, on l'utilise.
      matrice[p][r] = 1

    * la piece est > au reste.
      * on utilise la solution précédente en excluant la pièce
        matrice[p][r] = matrice[p - 1][r]

    * la pièce est < au reste,
      On choisit la meilleure des deux solutions :

      1. la solution précédente pour faire r, sans l'utiliser la pièce
      2. utiliser la précédente solution pour faire reste - valeur
        avec une pièce supplémentaire

        matrice[p][r] = min(matrice[p - 1][r],
                            1 + matrice[p][r - systeme[p - 1]])
```

   Correction

---

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**						
4								
5								
6								
7								

Correction

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	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

On y lit, par exemple que $\binom{7}{3} = 35$

Prenons l'exemple du nombre 20 (ligne 6). Il est obtenu en ajoutant les deux 10 de la ligne supérieure.

$\binom{6}{3} = \binom{5}{3} + \binom{5}{2} \iff 20 = 10 + 10$

Correction

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 $\binom{n}{k}$

   Correction Voici l'algorithme récursif qui correspond à ça :

```
fonction c(n, k):
  si k = 0 ou si k = n, alors retourner 1
  sinon retourner c(n-1, k) + c(n-1, k-1)
```

Correction

2. Construire l'arbre des appels récursifs du calcul de $\binom{5}{3}$

   (prévoir de la place, le nombre total d'appels récursifs est $2\binom{n}{k} - 2$).

   Correction

   Voici les appels récursifs de notre fonction simple pour calculer $\binom{5}{3}$ appels récursifs

   Correction

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 ?

   Correction Solution

   1. algorithme

      1. On remplit les k premières cases de chaque ligne de haut en bas.
      2. On arrête à la ligne n
      3. Temps: $T(n, k) = O(nk)$

   2. Seule l'avant dernière ligne doit être gardée en mémoire.

   3. Non, on peut améliorer en ne calculant que les cases "en haut à gauche" de la case initiale. **Correction**

---

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 $i_1, i_2, \cdots, i_k$ (avec $i_1 + i_2 + \cdots + i_k = n$) et obtenir comme profit la somme $p_{i1} + p_{i2} + \cdots + p_{ik}$ 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 ?

   Correction Solution

   - Aucune découpe: profit 9
   - Découpe 1 + 3: profit 1 + 8 = 9
   - Découpe 2 + 2: profit 5 + 5 = 10
   - Découpe 3 + 1: profit 8 + 1 = 9
   - Découpe 1+1+2: profit 1 + 1 + 5 = 7
   - Découpe 1+2+1: profit 1 + 5 + 1 = 7
   - Découpe 2+1+1: profit 5 + 1 + 1 = 7
   - Découpe 1+1+1+1: profit 1 + 1 + 1 + 1 = 4.
   - Optimal: découpe 2+2, c'est-à-dire 2 morceaux de longueur 2. **Correction**

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.

   Correction Solution

   recurrence 1

$$r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$$

- Le premier argument correspond p_n à ne faire aucune découpe
- les autres $n-1$ arguments, à faire un premier trait de découpe à la longueur i, et à obtenir le revenu maximal des deux morceaux obtenus de longueur i et $n-i$.

recurrence 2

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

en posant $r_0 = 0$.

S'obtient en considérant en que ce que l'on obtient au final est un morceau de longueur i sur l'extrémité gauche, au prix p_i, et le reste, de longueur $n-i$ qui, dans une solution optimale, doit être un découpage optimal d'une planche de longueur $n-i$. **Correction**

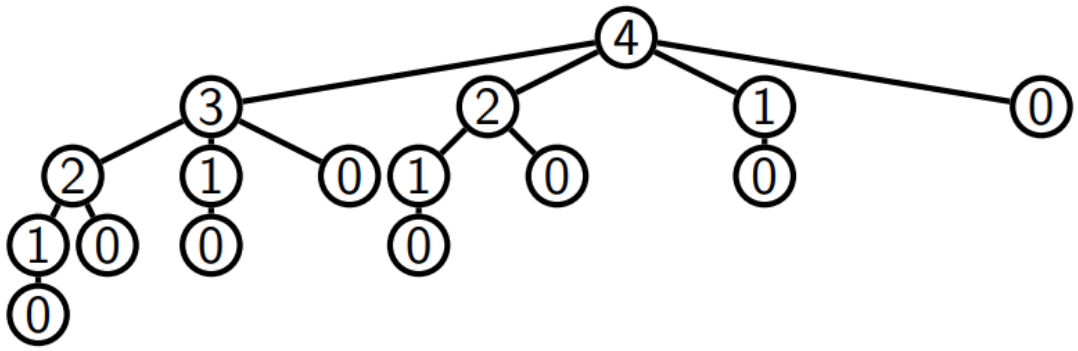
4. Proposer une fonction récursive `coupe` qui calcule r_n en fonction des paramètres p (le tableau des prix) et n (la longueur de la planche).

   Correction Solution

   `fonction coupe(p, n):  si n=0 alors retourner 0  sinon  q = - infini  pour i allant de 1 à n  q = max(q, p[i] + coupe(p, n-i))  retourner q` **Correction**

5. Construire l'arbre des appels récursifs de `coupe(p, 4)`

   Correction Solution



   Correction

6. Modifier votre fonction précédente pour qu'elle utilise une solution mémoisée.

   1. de haut en bas
   2. de bas en haut

   Correction

   1. On considère un tableau $r[0..n]$ dont tous les éléments sont initialement tels que $r[i] = -\infty$
   2. de haut en bas

```
Coupe_memoize(p,n,r):
  si r[n] >= 0 alors retourner r[n]
  sinon,
    si n = 0 alors q = 0
    sinon
      q = - infini
      pour i allant de 1 à n
        q = max(q, p[i] + Coupe_memoize(p, n-i, r))
```

```
    r[n] = q
  retourner q
```

Complexité $O(n^2)$

3. de bas en haut
   - On considère un tableau $r[0..n]$
   - $r[0] = 0$
   - pour i=1 à j
     - $q = -\infty$
     - pour i = 1 à j

       $q = max(q, p[i] + r[j - i])$
     - $r[j] = q$
   - retourner $r[n]$

À partir de r et s produits par l'algorithme plus haut, on peut reconstruire la solution. Il suffit de faire :

- tant que $n > 0$
  - afficher "un morceau de longueur $s[n]$"
  - $n = n - s[n]$
- sur l'exemple précédent on obtient :

longueur i	0	1	2	3	4	5	6	7	8	9	10
r_i	0	1	5	8	10	13	17	18	22	25	30
s_i	0	1	2	3	2	2	6	1	2	3	10

- Par exemple, pour $n = 7$ on affiche qu'il faut découper en un morceau de longueur 1 et un morceau de 6 (pour un profit de 18) **Correction**