Introduction à la programmation dynamique

I Principes généraux de la programmation dynamique

Afin de résoudre certains problèmes complexes, il existe différentes stratégies déjà rencontrée en première année :

- une stratégie *diviser pour mieux régner*, qui consiste à diviser un problèmes en plusieurs sous-problèmes indépendants : la recherche dichotomique dans une liste triée par exemple.
- une stratégie *gloutonne*, qui consiste à faire un choix localement optimal : le problème du rendu de monnaie dans lequel on choisit de commencer par utiliser la plus grosse pièce possible par exemple.

Chacune de ces deux stratégies possède des avantages et des défauts : les deux stratégies possèdent une complexité intéressante et sont simples à mettre en œuvre ; la stratégie diviser pour mieux régner demande de réussir à diviser le problème en sous problèmes indépendants et la stratégie gloutonne ne fournit pas toujours une solution optimale.

La *programmation dynamique* est une autre stratégie qui demande de diviser la résolution d'un problème en passant par des sous problèmes, mais, à la différence de la stratégie diviser pour mieux régner, ces sous problèmes peuvent ne pas être indépendants entre eux.

On a par exemple vu lors du calcul du $n^{\text{ème}}$ terme de la suite de Fibonacci (définie par $\forall n \in \mathbb{N}, f_{n+2} = f_{n+1} + f_n$) qu'il suffisait de calculer f_{n-1} et f_{n-2} (deux sous problèmes) pour déterminer la valeur de f_n. Comme le calcul de f_{n-1} nécessitait lui aussi de connaître la valeur de f_{n-2}, ces deux sous problèmes ne sont pas indépendants, ce qui pouvait rendre un calcul récursif naïf particulièrement inefficace (complexité exponentielle).

La programmation dynamique est une stratégie efficace lorsque l'on recherche à maximiser ou minimiser une certaine quantité. Plus particulièrement, on peut résoudre des problèmes qui présentent la propriété de **sous structure optimale** et qui nécessitent la gestion de **chevauchements** :

- on parle de **sous structure optimale** si la recherche de la solution optimale à un problème donné passe par la recherche (souvent récursive) de solutions optimales de sous problèmes.
- on parle de **chevauchement** des sous problèmes si ces sous problèmes ne sont pas indépendants et que leurs résolutions demandent de refaire plusieurs fois les mêmes calculs.

II Un cas concret de sous structure optimale

On dispose d'une matrice de $M = (m_{i,j}) \in \mathcal{M}_{n,p}(\mathbb{N})$ dont les coefficients sont des entiers naturels et on cherche à rejoindre la case en haut à gauche (indices $0, 0$) à celle en bas à droite (indice $n-1, p-1$) en respectant les règles suivantes :

— seuls les déplacements d'une case vers la droite ou vers le bas sont possibles,

— on cherche le chemin pour lequel la somme des coefficients rencontrés sera la plus grande possible (on parlera de chemin de récolte maximale).

La résolution du problème repose sur la propriété suivante : si le chemin de récolte maximale de $m_{0,0}$ à $m_{n-1,p-1}$ passe par la case d'indice i, j alors les chemins empruntés pour aller de $m_{0,0}$ à $m_{i,j}$ puis de $m_{i,j}$ à $m_{n-1,p-1}$ sont de récoltes maximales.

On peut donc déterminer la valeur de la récolte maximale de la façon suivante : si on note $r_{i,j}$ la récolte maximale dans un chemin allant de $m_{0,0}$ à $m_{i,j}$ alors on a

$$r_{i,j} = m_{i,j} + \begin{cases} 0 & \text{si } i=j=0 \\ r_{0,j-1} & \text{si } i=0 \quad \text{on suit obligatoirement le bord supérieur de } M \\ r_{i-1,0} & \text{si } j=0 \quad \text{on suit obligatoirement le bord gauche de } M \\ \max\{r_{i-1,j}, r_{i,j-1}\} & \text{si } i,j \geqslant 1 \quad \text{selon que le dernier déplacement se fait vers la droite ou vers le bas} \end{cases}$$

Les sous problèmes à résoudre pour déterminer $r_{n,p}$ sont donc : déterminer $r_{n-1,p}$ et $r_{n,p-1}$, qui sont donc deux solutions optimales à des sous problèmes mais qui ne sont pas indépendants.

Ces dernières formules peuvent amener à considérer une programmation récursive mais l'interdépendance des calculs de $r_{n-1,p}$ et $r_{n,p-1}$ rendrait la complexité exponentielle (au final très proche d'un calcul récursif naïf des coefficients binomiaux par la formule de Pascal).

III Résolution par programmation dynamique

1. Mémoïzation

Comme pour le calcul des coefficients binomiaux, il est possible de rendre le calcul récursif de $r_{n,p}$ en utilisant un dictionnaire :

```
def r(M) :
    n,p = len(M),len(M[0])
    dico = {}                    # mise en place du dictionnaire
    def r1(i,j,M) :              # une sous fonction qui complète le dictionnaire
        if (i,j) not in dico :
            if i == j == 0 :
                s = M[0][0]
            elif i == 0 :
                s = r1(0,j-1,M) + M[0][j]
            elif j == 0 :
                s = r1(i-1,0,M) + M[i][0]
            else :
                s = M[i][j] + max([r1(i-1,j,M),r1(i,j-1,M)])
            dico[(i,j)] = s
        return dico[(i,j)]
    return r1(n-1,p-1,M)         # appel à la sous fonction
```

Cette fonction, relativement simple, permet de déterminer de façon efficace la récolte maximale mais rend peu évident la détermination de sa complexité. Il est également difficile de déterminer le chemin permettant d'obtenir cette récole maximale.

2. De bas en haut

On peut également faire le calcul en utilisant une autre matrice $R = (r_{i,j}) \in \mathcal{M}_{n,p}(\mathbb{N})$ (une liste de listes plus exactement) de façon à mémoriser toutes les valeurs de $r_{i,j}$: un avantage de cette méthode est de pouvoir faire un calcul de complexité très simplement.

```
def R(M) :
    n,p = len(M),len(M[0])
    B = [[0 for j in range(p)] for i in range(n)]
    B[0][0] = M[0][0]
    # première colonne
    for i in range(1,n) :
        B[i][0] = B[i-1][0] + M[i][0]
    # première ligne
    for j in range(1,p) :
        B[0][j] = B[0][j-1] + M[0][j]
    for i in range(1,n) : # le reste
        for j in range(1,p) :
            B[i][j] = M[i][j] + max([B[i-1][j],B[i][j-1]])
    return B
```

$$M = \begin{pmatrix} 2 & 1 & 7 & 2 \\ 3 & 1 & 3 & 3 \\ 5 & 1 & 2 & 4 \end{pmatrix} \quad \text{donne} \quad R = \begin{pmatrix} 2 & 3 & 10 & 12 \\ 5 & 6 & 13 & 16 \\ 10 & 11 & 15 & 20 \end{pmatrix}$$

La récolte maximale cherchée est la valeur $r_{2,3} = 20$ dans cet exemple.
Si on souhaite que la fonction précédente ne renvoie que la valeur de cette récolte maximale, on remplace la dernière ligne par `return B[-1][-1]`.

La complexité de cette fonction est déterminée de la façon suivante :

- l'initialisation de la « matrice » B est en $O(np)$
- le remplissage de la première colonne est en $O(n)$, celui de la première ligne en $O(p)$
- le reste du remplissage se fait avec deux boucles imbriquées et est donc en $O(np)$.

La complexité de cette fonction est donc en $O(np)$.

Un autre avantage de cette matrice est de pouvoir l'utiliser pour déterminer aussi le chemin permettant cette récolte maximale : en partant de la case finale, on remonte à la case initiale (d'indices $i = j = 0$) en retrouvant les choix qui avaient été faits.

```
def chemin(M) :
    i,j = len(M)-1,len(M[0])-1          # case finale
    L = [(i,j)]
    B = R(M)                            # on récupère toute la matrice précédente
    while i>0 and j>0 :
        if B[i-1][j] > B[i][j-1] :      # on venait d'en haut
            i -= 1
        else :                          # ou de la gauche
            j -=1
        L.append((i,j))
    L = L[::-1]                         # on remet L dans le bon ordre
    # si on a atteint le bord supérieur, on concatène à gauche puisque c'est
      le début du chemin
    if i == 0 :
        L = [(0,k) for k in range(j)] + L
    # ou le bord gauche
    else :
        L = [(k,0) for k in range(i)] + L
    return L
```

Ce qui donne `[(0, 0), (0, 1), (0, 2), (1, 2), (1, 3), (2, 3)]` et qui correspond au chemin suivant :

$$M = \begin{pmatrix} \mathbf{2} & \mathbf{1} & \mathbf{7} & 2 \\ 3 & 1 & \mathbf{3} & \mathbf{3} \\ 5 & 1 & 2 & \mathbf{4} \end{pmatrix}$$

Pour cette fonction, la complexité est à nouveau en $O(np)$, notamment à cause de l'utilisation de la fonction `R(M)` dont la complexité est déjà en $O(np)$. La reconstruction du chemin possède par contre une complexité linéaire en $n + p$ donc n'a pas une grosse incidence sur la complexité finale.

IV Comparaison des différentes stratégies

1. Force brute ou recherche exhaustive

Une méthode non évoquée jusqu'ici serait de tester tous les chemins possibles : comme un chemin est constitué de $n-1$ déplacements vers la droite et $p-1$ vers le bas, le nombre de chemins à étudier est le nombre de façon de choisir la position de $p-1$ déplacement vers le bas parmi les $n+p-2$ déplacements donc $\binom{n+p-2}{p-1}$, ce qui conduirait à une complexité déraisonnable.

2. Stratégie gloutonne

Le code de la stratégie gloutonne est le suivant :

```
def glouton(M) :
    n,p = len(M),len(M[0])
    i,j = 0,0
    s = M[0][0]
    while i+j < n+p-2 :
        if i == n-1 :                    # on est déjà sur le bord du bas
            j += 1
        elif j == p-1 :                  # ou de droite
            i += 1
        elif M[i+1][j] > M[i][j+1] :     # récolte plus grande vers le bas
            i += 1
        else :
            j += 1
        s += M[i][j]
    return s
```

La complexité devient linéaire en $n+p$ mais la recette trouvée (17) n'est pas optimale ; elle correspond au chemin suivant :

$$M = \begin{pmatrix} \mathbf{2} & 1 & 7 & 2 \\ \mathbf{3} & 1 & 3 & 3 \\ \mathbf{5} & \mathbf{1} & \mathbf{2} & \mathbf{4} \end{pmatrix}$$

3. Programmation dynamique

La complexité du code de bas en haut est $O(np)$ donc moins bonne que l'approche gloutonne mais fournit bien une solution optimale. La programmation dynamique possède aussi une complexité spatiale plus importante à cause de l'utilisation de la matrice R (ou du dictionnaire).