Chapitre 7

Programmation dynamique

La programmation dynamique est une méthode de construction de solution à un problème consistant à réduire la résolution sur une entrée de taille N à la résolution du même problème sur des entrées de taille $< N$. C'est un paradigme de programmation pour la résolution de problèmes difficiles.

1 Principe de programmation dynamique

1.1 Exemple du calcul récursif des coefficients binomiaux

Essayons d'écrire une fonction efficace pour le calcul d'un coefficient binomial $\binom{n}{p}$. Appliquer la formule en factorielle occasionnerait des multiplications nombreuses sur des grandeurs de grande taille et risquerait d'occasionner rapidement des erreurs de dépassement de capacité. On optera plutôt pour l'usage de la relation de Pascal :

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

agrémentée de valeurs initiales :

$$\forall n \in \mathbb{N},\ \binom{n}{0} = \binom{n}{n} = 1$$

pour obtenir le code récursif :

```
def coefbinomiaux(n,p):
    if p < 0 or p > n:
        return 0
    if p == 0 or p == n:
        return 1
    return coefbinomiaux(n-1,p)+coefbinomiaux(n-1,p-1)
```

Même si la solution est élégante, et fonctionne parfaitement pour de petites valeurs de n et p, elle s'avère très rapidement impraticable, pour des valeurs relativement peu élevées de n et p :

```
In [2]: coefbinomiaux(3,2)
Out[2]: 3

In [3]: coefbinomiaux(6,2)
Out[3]: 15

In [4]: %timeit coefbinomiaux(20,10)
10 loops, best of 3: 128 ms per loop

In [5]: %timeit coefbinomiaux(30,15)
1 loops, best of 3: 1min 44s per loop
```

On comprend aisément, à la lueur d'exemple déja rencontrés comme le calcul récursif de la suite de fibonacci (cf. Chapitre récursivité), que les deux appels en fin de fonction contribuent à une complexité exponentielle.

Notons $C(n,p)$ l'ordre du nombre d'opérations effectuées à l'appel avec pour paramètres (n,p). Alors :

$$C(n,0) = C(n,n) = 1 \quad \text{et} \quad \forall p \in [\![1; n-1]\!], C(n,p) = C(n-1,p-1) + C(n-1,p) + 1$$

Une récurrence immédiate (avec la relation de Pascal) montre alors que :

$$C(n,p) = \binom{n}{p} - 1$$

La formule de Stirling, permet d'établir par exemple pour le calcul de $\binom{2n}{n}$ une complexité exponentielle de l'ordre de $\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}}$.

Nous avons vu à l'occasion comment contourner la difficulté : les coefficients binomiaux $\binom{n}{p}$ pour n et p petits sont recalculés inutilement un très grand nombre de fois. Il suffit de les mémoriser dans un conteneur afin de ne pas les recalculer inutilement à chaque appel. On utilise en général un dictionnaire, et on appelle cela la mémoisation.

```
def coefbin_rec(n,p,Dic):
    if (n,p) in Dic:
        return Dic[(n,p)]
    if p < 0 or p > n:
        return 0
    if p == 0 or p == n:
        return 1
    Dic[(n,p)] = coefbin_rec(n-1,p,Dic)+coefbin_rec(n-1,p-1,Dic)
    return Dic[(n,p)]

def coefbinomiaux(n,p):
    Dic = {}
    return coefbin_rec(n,p,Dic)
```

La fonction devient très efficace :

```
In [7]: coefbinomiaux(30,15)
Out[7]: 155117520

In [8]: %timeit coefbinomiaux(30,15)
1000 loops, best of 3: 261 µs per loop
```

1.2 Principe de la programmation dynamique

On considère un problème, assimilé au calcul de l'image par une fonction f de paramètres $p_1, \dots, p_n$.

1. On cherche à établir une relation de récurrence qui exprime $f(p_1, \dots, p_n)$ à l'aide d'image par f de paramètres $q_1, \dots, q_n$ inférieurs, c'est à dire que pour tout $i \in [\![1, n]\!]$, $q_i \leqslant p_i$ et en garantissant la terminaison. C'est une condition préliminaire caractéristique à l'usage d'une programmation dynamique.
2. On résout ces sous-problèmes, soit de bas en haut par itération, soit de haut en bas par récursivité en les stockant, par mémoisation (à l'aide d'un dictionnaire).
3. On en déduit $f(p_1, \dots, p_n)$ à l'aide de la relation de récurrence.

La programmation dynamique est souvent utilisée pour résoudre des problèmes d'optimisation combinatoire, c'est à dire consistant à déterminer le maximum ou minimum d'une fonction

$$\max_{(p_1,\dots,p_n)\in D} f(p_1, \dots, p_n)$$

dépendant de paramètres $p_1, \dots, p_n$ décrivant un domaine discret D.

Le principe s'applique à la condition que le problème satisfasse le principe d'optimalité de Bellman : à savoir que les sous-solutions d'une solution optimale soient aussi optimales. On parle aussi de sous-structure optimale.
Par exemple pour la recherche de géodésique dans un graphe, tout sous-chemin d'une géodésique est aussi une géodésique.
La mémoisation est utile pour contourner le problème de chevauchement des sous-problèmes et éviter de recalculer plusieurs fois des résolutions intermédiaires. Elle peut permettre aussi de déterminer la solution optimale après avoir déterminé la valeur optimale.

La résolution peut se faire soit :

– de haut en bas, en stockant bien les résolutions intermédiaires par mémoisation afin que la complexité n'explose pas.

– de bas en haut, en construisant progressivement la solution optimale. On utilise en général dans ce cas un tableau pour stocker les résultats intermédiaires.

Le mieux est encore de regarder quelques exemples.

L'exemple précédent du calcul des coefficients binomiaux procédait de haut en bas et avec memoisation.

Dans les exemples qui suivent on procède de bas en haut en stockant les résultats intermédiaires au sein d'un tableau.

2 Exemples de programmation dynamique

2.1 Chemin de poids maximal dans une matrice

On se donne une matrice $(a_{i,j})_{\substack{0\leqslant i\leqslant n\\ 0\leqslant j\leqslant m}}$ dont les coefficients sont des entiers positifs. On cherche un chemin partant de la case en haut à gauche indicé $(0, 0)$ et aboutissant à la case en bas à droite (n, m) en ne procédant qu'a des déplacements de gauche à droite ou de haut en bas, d'un cran, et dont la somme des coefficients (le poids) soit maximale.

On pourrait bien sûr calculer les poids de tous les chemins possibles, puisqu'ils sont en nombre fini : $\binom{n+m-2}{n-1}$ (il suffit de savoir où placer les $n-1$ déplacement en bas parmi les $n-1+m-1 = n+m-2$ déplacement nécessaires). Pour $n = m$ l'application de Stirling nous donne un nombre de chemins à analyser équivalent à $\frac{4^{n-1}}{\sqrt{\pi n}}$, soit exponentiel. C'est donc irréalisable en pratique pour des valeurs de n et m pas trop petites.

Une solution optimale au problème consisterait en un chemin de la case $(0, 0)$ à la case $(n-1, m-1)$ de poids maximal. S'il passe par la case (i, j), le sous-chemin initial de

$(0,0)$ à (i,j) est aussi de poids maximal parmi tous les chemins de $(0,0)$ à (i,j) ; en effet s'il ne l'était pas on pourrait en déduire un chemin de poids strictement inférieur reliant $(0,0)$ à $(n-1,m-1)$. On vérifie donc une propriété de sous-structure optimale.
Notons $p_{i,j}$ le poids maximal d'une chemin de $(0,0)$ à la case (i,j). Alors :

$$p_{i,j} = a_{i,j} + \begin{cases} 0 & \text{si } i=j=0 \\ p_{i,j-1} & \text{si } i=0 \text{ et } j>0 \\ p_{i-1,j} & \text{si } i>0 \text{ et } j=0 \\ \max(p_{i-1,j}, p_{i,j-1}) & \text{sinon} \end{cases}$$

Les trois premiers points sont évidents ; ils correspondent aux cas limites où la case est située sur le bord gauche ou haut de la matrice et dans ce cas un chemin est unique, et donc de poids maximal. Quant à la dernière, le poids maximal est obtenu en ajoutant le coefficient de la case (i,j) au plus grand des deux poids maximaux des chemins menant à la case du dessus $(i,j-1)$ ou à la case à gauche $(i-1,j)$; car un sous-chemin d'un chemin de poids maximal est lui même de poids maximal.
L'idée consiste alors à remplir une matrice de même taille (n,m) avec les poids maximaux $p_{i,j}$ pour tout $(i,j) \in [\![0,n-1]\!] \times [\![0,m-1]\!]$, en partant du coin gauche et en appliquant la relation de récurrence ci-dessus. Le code Python obtenu est :

```
def matricePoids(A):
    n, m = len(A), len(A[0])
    M = [ [0] * m for k in range(n) ]
    M[0][0] = A[0][0]          # poids (0,0)
    for i in range(1,n):       # poids (i,0) bord gauche
        M[i][0] = M[i-1][0] + A[i][0]
    for j in range(1,m):       # poids (0,j) bord haut
        M[0][j] = M[0][j-1] + A[0][j]
    for i in range(1,n):
        for j in range(1,m):
            M[i][j] = A[i][j] + max(M[i-1][j],M[i][j-1])
    return M
```

Par exemple :

```
In [10]:  A =  [[1,2,1,3,2,1],
                [2,1,1,1,2,2],
                [1,2,1,3,1,3],
                [1,1,2,1,3,1]]
In [11]:  matricePoids(A)
Out[11]:
[[1, 3, 4, 7, 9, 10],
 [3, 4, 5, 8, 11, 13],
 [4, 6, 7, 11, 12, 16],
 [5, 7, 9, 12, 15, 17]]
```

$$\begin{pmatrix} 1 & 2 & 1 & 3 & 2 & 1 \\ 2 & 1 & 1 & 1 & 2 & 2 \\ 1 & 2 & 1 & 3 & 1 & 3 \\ 1 & 1 & 2 & 1 & 3 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} \underline{1} & \underline{3} & \underline{4} & \underline{7} & \underline{9} & 10 \\ 3 & 4 & 5 & 8 & \underline{11} & \underline{13} \\ 4 & 6 & 7 & 11 & 12 & \underline{16} \\ 5 & 7 & 9 & 12 & 15 & \underline{17} \end{pmatrix}$$

Le poids maximal est ici 17, réalisé par le chemin souligné en rouge. Il serait facile d'écrire un script le déduisant de la matrice renvoyée par la fonction `matricePoids`. Il suffirait en partant de la case en bas à droite de remonter en choisissant entre les cases du dessus et à gauche celle de poids maximal (dès que les deux poids sont égaux, le nombre de chemins de poids maximal est doublé).
Cet algorithme a une complexité spatiale et temporelle en $O(nm)$.

2.2 Plus longue sous-séquence commune

Soit $s = s_1 s_2 \cdots s_n$ une chaine de caractère, suite des caractères $s_1, s_2, \ldots, s_n$; on appelle sous-séquence de s toute chaine de caractère $s_{\sigma(1)} s_{\sigma(2)} \cdots s_{\sigma(p)}$ où $\sigma = [\![1,p]\!] \longrightarrow [\![1,n]\!]$ est une fonction strictement croissante.

Données deux chaines de caractère, un problème récurrent consiste à déterminer une plus longue sous-séquence commune. Par exemple dans “epithete” et “pythagore”, la sous-séquence commune “pthe” est la plus longue.

Ce problème prend des applications importantes, notamment en phylogénétique.

Soient $s = s_1 \cdots s_n$ et $t = t_1 \cdots t_m$ deux chaines de caractères de longueurs respectives n et m. Pour tout $(i,j) \in [\![0,n]\!] \times [\![0,m]\!]$ notons $s[1,i] = s_1 \cdots s_i$, $t[1,j] = t_1 \cdots t_j$ les préfixes de s et t de longueurs respectives i et j, et $\ell_{i,j}$ la longueur d'une plus longue sous-séquence commune à $s[1,i]$ et $t[1,j]$.

On recherche $\ell_{n,m}$ et une plus longue sous-séquence commune ω de longueur maximale $\ell_{n,m}$. On remarque que :

- Si s et t ont même dernier caractère, alors ω termine nécessairement par ce caractère. En notant ω', s' et t' le préfixe de ω, s et t obtenu en supprimant leur dernier caractère, alors ω' est une sous-séquence commune et de longueur maximale à s' et t'.
- Si s et t n'ont pas même dernier caractère, alors avec les mêmes notations ω est une sous-séquence commune et de longueur maximale à s et t', ou à s' et t (voire aux deux).

Ceci procure une sous-structure optimale. On obtient alors la relation de récurrence sur les $\ell_{i,j}$:

$$\ell_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ 1 + \ell_{i-1,j-1} & \text{si } s[1,i] \text{ et } t[1,j] \text{ finissent par le même caractère} \\ \max(\ell_{i-1,j}, \ell_{i,j-1}) & \text{sinon} \end{cases}$$

L'idée va alors consister à calculer itérativement la famille des $\ell_{i,j}$ à l'aide de cette relation de récurence au sein d'une matrice de taille $(n+1) \times (m+1)$, l'élément $\ell_{i,j}$ se trouvant finalement à la position (i,j). On obtiendra alors la longueur maximale $\ell_n m$ recherchée à la position (n,m) de cette matrice, et on reconstruira une sous-séquence maximale en remontant de (n,m) à $(0,0)$ de la façon suivante : en partant de la chaine vide `c=""` on remonte :

- de la position (i,j) à la position $(i-1,j)$ si $\ell_{i,j} = \ell_{i-1,j}$,
- de la position (i,j) à la position $(i,j-1)$ si $\ell_{i,j} = \ell_{i,j-1}$,
- de la position (i,j) à la position $(i-1,j-1)$ sinon, et on ajoute le dernier caractère de $s[1,i]$ (ou de $t[1,j]$) en début de la chaine c.

```
def plusLongueSsSeq(s,t):
    n, m = len(s), len(t)
    M = [[0] * (m+1) for k in range(n+1)]
    # Remplissage matrice M
    for i in range(1,n+1):
        for j in range(1,m+1):
            if s[i-1] == t[j-1]:
                M[i][j] = M[i-1][j-1] + 1
            else:
                M[i][j] = max(M[i-1][j], M[i][j-1])
    ell = M[n][m]               # longueur maximale
    c = ""                      # ss-sequence max à constituer
    i, j = n, m                 # On part d'en bas à droite
    while len(c) < ell:
        if M[i][j] == M[i-1][j]:
            i = i-1             # en haut
        elif M[i][j] == M[i][j-1]:
            j = j-1             # à gauche
        else:                   # caractere commun à ajouter
            i = i-1
            j = j-1
            c = s[i] + c        # ajout caractere
    return ell, c               # renvoi Lgr max et SsSeq max
```

Exemple :

```
In [12]: plusLongueSsSeq("epithete","pythagore")

Out[12]: (4, 'pthe')
```

La complexité spatiale et temporelle est encore en $O(nm)$.

2.3 Distance d'édition

C'est un problème important en génétique ; il consiste à déterminer le nombre minimal d'édition permettant de passer d'une chaine de caractère donnée à une autre. Un édition consiste en :

- L'insertion d'un caractère à une position donnée,
- La suppression d'une caractère,
- La modification d'un caractère.

Par exemple : on peut montrer que la distance d'édition entre les chaines $s =$ "$ACGTCA$" et $t =$ "$CACGTG$" vaut 3 ; une suite de 3 éditions changeant la première en la deuxième est donnée par :

$$ACGTCA \underset{\text{suppression}}{\longrightarrow} ACGTC \underset{\text{modification}}{\longrightarrow} ACGTG \underset{\text{insertion}}{\longrightarrow} CACGTG$$

Cela définit une distance $d(s,t)$ au sens mathématique entre des chaines de caractères s et t ; c'est à dire vérifiant les propriétés suivantes :

- $d(s,s) = 0$
- $d(s,t) = d(t,s)$
- $d(s,t) \leqslant d(s,u) + d(u,t)$ pour toute chaine u (inégalité triangulaire)

Une sous-structure optimale s'observe facilement (comme souvent dans le cas de distances) : si une suite d'édition de longueur minimale permet de passer de la chaine s à la chaine t en passant par une chaine u ($s \longrightarrow u \longrightarrow t$) alors les sous-suites d'édition $s \longrightarrow u$ et $u \longrightarrow t$ sont aussi de longueurs minimales.

Comme dans l'exemple précédent, pour des chaines $s = s_1 \cdots s_n$ et $t = t_1 \cdots t_m$ de longueur respective n et m, notons pour tout $i \in [\![0,n]\!]$ et $j \in [\![0,m]\!]$, $s[1,i]$ et $t[1,j]$ les préfixes de s et t de longueur respective i et j. Notons $d_{i,j}$ la distance d'édition entre

$s[1,i]$ et $t[1,j]$. On a alors les relations de récurrence :

$$d_{i,j} = \begin{cases} i & \text{si } j = 0 \\ j & \text{si } i = 0 \\ d_{i-1,j-1} & \text{si } s_i = t_j \\ 1 + \min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) & \text{sinon} \end{cases}$$

Les 3 premières sont évidentes : montrons la dernière.

On est dans la situation où $s[1,i]$ et $t[1,j]$ sont non vides et n'ont pas même dernier caractère.

Tout d'abord $d_{i,j} \leqslant 1 + \min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1})$. En effet : $d_{i,j} \leqslant 1 + d_{i-1,j}$ car si on peut passer de $s[1,i-1]$ à $t[1,j]$ par une suite de $d_{i-1,j}$ éditions, une édition supplémentaire consistant initialement à supprimer le dernier caractère de $s[1,i]$ avant d'appliquer la suite d'éditions précédente précédente de $s[1,i-1]$ à $t[1,j]$ permettra de passer de $s[1,i]$ à $t[1,j]$ en $1 + d_{i-1,j}$ éditions.

Le même argument montre qu'on a aussi $d_{i,j} \leqslant 1 + d_{i,j-1}$ (en commençant par supprimer le dernier caractère de $t[1,j]$) et $d_{i,j} \leqslant 1 + d_{i-1,j-1}$ (en commençant par modifier le dernier caractère de $s[1,i]$). D'où l'inégalité $d_{i,j} \leqslant 1 + \min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1})$.

Montrons l'inégalité inverse $d_{i,j} \geqslant 1 + \min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1})$. Les chaines $s[1,i]$ et $t[1,j]$ n'ayant pas le même dernier caractère, dans une suite d'éditions de longueur minimal $d_{i,j}$ transformant $s[1,i]$ en $t[1,j]$ ce dernier caractère est apparu à sa place définitive dans $t[1,j]$ après une insertion, modification ou suppression. Une suite d'éditions optimale peut être modifiée pour que cette dernière édition sur le dernier caractère de $t[1,j]$ soit effectuée en dernier.

Si par exemple c'est une insertion, supprimer cette dernière étape permet de passer de $s[1,i]$ à $t[1,j-1]$ par une suite d'édition de longueur $d_{i,j} - 1$; ainsi on aurait

$$\min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) \leqslant d_{i,j-1} \leqslant d_{i,j} - 1$$

Si c'est une suppression, supprimer cette dernière étape permet de passer de $s[1,i-1]$ à $t[1,j]$ par une suite d'édition de longueur $d_{i,j} - 1$; ainsi on aurait

$$\min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) \leqslant d_{i-1,j} \leqslant d_{i,j} - 1$$

Enfin si c'est une modification, on aurait

$$\min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) \leqslant d_{i-1,j-1} \leqslant d_{i,j} - 1$$

ce qui montre dans les 3 cas que :

$$1 + \min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) \leqslant d_{i,j}$$

On obtient le code donnant la distance d'édition :

```
def distEdition(s,t):
    n, m = len(s), len(t)
    D = [[0] * (m+1) for k in range(n+1)]
    for i in range(1,n+1):
        D[i][0]=i
    for j in range(1,m+1):
        D[0][j] = j
    for i in range(1,n+1):
        for j in range(1,m+1):
            if s[i-1] == t[j-1]:
                D[i][j] = D[i-1][j-1]
            else:
                D[i][j] = 1+min(D[i-1][j],D[i][j-1],D[i-1][j-1])
    return D[n][m]
```

Par exemple, pour les mots "ACGTCA" et "CACGTG" comme ci-dessus, on obtient bien une distance d'édition de 3 ; la matrice constituée est :

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 1 & 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 2 & 1 & 2 & 3 & 4 \\ 3 & 2 & 2 & 2 & 1 & 2 & 3 \\ 4 & 3 & 3 & 3 & 2 & 1 & 2 \\ 5 & 4 & 4 & 3 & 3 & 2 & 2 \\ 6 & 5 & 4 & 4 & 4 & 3 & 3 \end{pmatrix}$$