Programmation dynamique II

Fabrice Lembrez - PSI*

Lycée Pierre de Fermat

Nous avons vu que certains problèmes d'optimisation se prêtent à un calcul récursif via les équations de Bellmann.

Il reste à voir

- Comment résoudre ces équations et trouver la valeur du maximum
- Comment obtenir la configuration qui donne ce maximum

Plan

Problème : calcul des coefficients

On souhaite calculer un coefficient binomial ($0 \leq p \leq n$) **en n'utilisant que des sommes**, et donc employer la formule de Pascal qu'on notera ici

$$B(n,p) = B(n-1,p-1) + B(n-1,p), \qquad B(n,0) = B(n,n) = 1$$

Solution naïve

Une solution naïve est d'imiter l'écriture mathématique, mais cela ne permet pas d'arriver à des valeurs elles que $n = 30$:

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

Pourquoi c'est inefficace : il y a en gros croissance exponentielle du nombre d'appels (duplications successives $\rightarrow 2^n$). C'est une redondance inutile puisqu'on recalcule de nombreuses fois les mêmes quantités.

Solution naïve

Une solution naïve est d'imiter l'écriture mathématique, mais cela ne permet pas d'arriver à des valeurs elles que $n = 30$:

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

Pourquoi c'est inefficace : il y a en gros croissance exponentielle du nombre d'appels (duplications successives $\rightarrow 2^n$). C'est une redondance inutile puisqu'on recalcule de nombreuses fois les mêmes quantités.

Objectif prioritaire : complexité temporelle

On vise une programmation efficace en complexité temporelle, quitte à créer de la complexité spatiale.

Plan

Calcul de bas en haut (non récursif)

On peut suivre une démarche séquentielle "naturelle" en remplissant le triangle de Pascal ligne par ligne. On calcule donc tous les coefficients nécessaires dans une liste de listes un rectangle donc même si on ne calcule que les coefficients dans le triangle.

```
def BinomBasEnHaut(n,p):
    bino=[[0 for i in range(p+1)] for j in range(n+1)
    for i in range(n+1):
        bino[i][0]=1
        for j in range(1,min(i,p)+1):
            bino[i][j]=bino[i-1][j]+bino[i-1][j-1]
    return bino[n][p]
```

Complexité

- temporelle $O(np)$
- spatiale $O(np)$ aussi, même si on peut optimiser : on pourrait utiliser deux lignes et les écraser à chaque étape

Plan

Mémoïsation

Déf - Mémoïsation

C'est le stockage des valeurs renvoyées par une fonction (à l'aide d'une structure adaptée), pour éviter des appels superflus.
Elle se fait souvent à l'aide d'un dictionnaire.

Pourquoi un dictionnaire ? Il faut pouvoir répondre rapidement à la question : cette valeur a-t-elle été appelée ? et si oui, avec quel retour ?

Par exemple, pour le coefficient binomial, puisque les tuples sont non mutables, ils sont hachables, on appellera donc les valeurs via `dicoB[(n,p)]`.

Code de mémoïsation

```
def remplitDico(n,p,M):
    if p==0:
        M[(n,p)]=1
    elif n==0: #et p>0
        M[(n,p)]=0
    else:
        if (n-1,p-1) not in M:
            remplitDico(n-1,p-1,M)
        if (n-1,p) not in M:
            remplitDico(n-1,p,M)
        M[(n,p)]=M[(n-1,p-1)]+M[(n-1,p)]

def BinomHautEnBas(n,p):
    M=dict()
    remplitDico(n,p,M)
    return M[(n,p)]
```

Exemple : `BinomHautEnBas(7,5,dicoB)`

Performance

Ainsi si on demande `BinomHautEnBas(7,5,dicoB)` à partir d'un dictionnaire vide, on trouve bien 42 et le contenu du dictionnaire est

```
{(2, 1):2, (3, 2):3, (4, 3):4, (5, 4):5, (6, 5):6, (3, 1):3,
           (4, 2):6, (5, 3):10, (6, 4):15, (7, 5):21}
```

Seules les valeurs utiles ont été appelées. Et elles sont en mémoire si on calcule un autre coefficient binomial.

Conseil

Déjà retenir que le "bas" c'est les "petits indices".

On évite les ennuis

Si on vous fait établir une relation de récurrence, sauf demande expresse du contraire, faites un remplissage de bas en haut.

Plan

Rappel du problème : distance de Levenshtein

Distance = nombre minimal de suppressions et d'insertions pour passer d'un texte à un autre.

Version dynamique

$M[i][j]$: la distance d'édition pour passer de $\mathtt{t}_1[0:i]$ à $\mathtt{t}_2[0:j]$.

Rappel du problème : distance de Levenshtein

Distance = nombre minimal de suppressions et d'insertions pour passer d'un texte à un autre.

Version dynamique

$M[i][j]$: la distance d'édition pour passer de $\texttt{t}_1\texttt{[0:}i\texttt{]}$ à $\texttt{t}_2\texttt{[0:}j\texttt{]}$.

Equation de Bellmann : pour i,j fixé, on regarde ce qui arrive au dernier caractère d'une solution optimale

- si `t1[i]==t2[j]` on ne lui fait rien, puis `t1[:i-1]`→`t2[:j-1]`.
- sinon insertion de `t2[j]` puis `t1[:i]`→`t2[:j-1]`.
- sinon délétion de `t1[i]` puis `t1[:i-1]`→`t2[:j]`.

$$\texttt{M[i][j]} = \begin{cases} \texttt{M[i-1][j-1]} & \text{si } \texttt{t1[i]==t2[j]}, \\ \min(\texttt{1+M[i][j-1]}, \texttt{1+M[i-1][j]}) & \text{sinon} \end{cases}$$

Rappel du problème : distance de Levenshtein

Distance = nombre minimal de suppressions et d'insertions pour passer d'un texte à un autre.

Version dynamique

$M[i][j]$: la distance d'édition pour passer de $\texttt{t}_1\texttt{[0:}i\texttt{]}$ à $\texttt{t}_2\texttt{[0:}j\texttt{]}$.

Equation de Bellmann : pour i,j fixé, on regarde ce qui arrive au dernier caractère d'une solution optimale
- si `t1[i]==t2[j]` on ne lui fait rien, puis `t1[:i-1]`→`t2[:j-1]`.
- sinon insertion de `t2[j]` puis `t1[:i]`→`t2[:j-1]`.
- sinon délétion de `t1[i]` puis `t1[:i-1]`→`t2[:j]`.

$$\texttt{M[i][j]} = \begin{cases} \texttt{M[i-1][j-1]} & \text{si } \texttt{t1[i]==t2[j]}, \\ \min(\texttt{1+M[i][j-1]}, \texttt{1+M[i-1][j]}) & \text{sinon} \end{cases}$$

Meilleur procédé de remplissage : de bas en haut (= non récursif).

Analyser le résultat

Quand on fait de la programmation dynamique on résout plusieurs problèmes d'optimisation à la fois MAIS ils ne sont pas tous utiles pour le problème effectivement étudié

		U	A	B	C	C	X	Y	Z
	0	1	2	3	4	5	6	7	8
A	1	2	1	2	3	4	5	6	7
B	2	3	2	1	2	3	4	5	6
C	3	4	3	2	1	2	3	4	5
D	4	5	4	3	2	3	4	5	6
C	5	6	5	4	3	2	3	4	5
E	6	7	6	5	4	3	4	5	6
F	7	8	7	6	5	4	5	6	7

Question : quel chemin suivre pour atteindre la case finale ? On le trouve forcément en remontant.

La remontée

Rappel

$$\texttt{M[i][j]} = \begin{cases} \texttt{M[i-1][j-1]} & \text{si } \texttt{t1[i]==t2[j]}, \\ \min(\texttt{1+M[i][j-1]}, \texttt{1+M[i-1][j]}) & \text{sinon} \end{cases}$$

		U	A	B	C	C	X	Y	Z
	0	1	2	3	4	5	6	7	8
A	1	2	1	2	3	4	5	6	7
B	2	3	2	1	2	3	4	5	6
C	3	4	3	2	1	2	3	4	5
D	4	5	4	3	2	3	4	5	6
C	5	6	5	4	3	2	3	4	5
E	6	7	6	5	4	3	4	5	6
F	7	8	7	6	5	4	5	6	7

- Initialiser :
  - `chaine=t2`
  - `i=len(t1), j=len(t2)`
- quand `t1[i-1]=t2[j-1]`, décrémenter i et j
- sinon chercher qui a donné le min
  - une insertion ?
  - une délétion ?
- et appliquer l'opération inverse sur `chaine`

La remontée

Rappel

$$\texttt{M[i][j]} = \begin{cases} \texttt{M[i-1][j-1]} & \text{si } \texttt{t1[i]==t2[j]}, \\ \min(\texttt{1+M[i][j-1]}, \texttt{1+M[i-1][j]}) & \text{sinon} \end{cases}$$

		U	A	B	C	C	X	Y	Z
	0	1	2	3	4	5	6	7	8
A	1	2	1	2	3	4	5	6	7
B	2	3	2	1	2	3	4	5	6
C	3	4	3	2	1	2	3	4	5
D	4	5	4	3	2	3	4	5	6
C	5	6	5	4	3	2	3	4	5
E	6	7	6	5	4	3	4	5	6
F	7	8	7	6	5	4	5	6	7

- Initialiser :
  - `chaine=t2`
  - `i=len(t1), j=len(t2)`
- quand `t1[i-1]=t2[j-1]`, décrémenter i et j
- sinon chercher qui a donné le min
  - une insertion ?
  - une délétion ?
- et appliquer l'opération inverse sur `chaine`

On obtient ainsi une solution "retournée"

Plan

Le rendu de monnaie

Comment découper une somme S de façon optimale si on a des pièces de 1, des billets de 7 et 23 ? On a vu une formule de récurrence

$$N_S = \begin{cases} \min(N_{S-1}, N_{S-7}, N_{S-23}) + 1 & \text{si } S \geq 23 \\ \min(N_{S-1}, N_{S-7}) + 1 & \text{si } S \geq 7 \\ N_{S-1} + 1 & \text{sinon} \end{cases}$$

Qui se justifie en regardant la dernière pièce rendue.

On peut suivre la même démarche : retrouver la dernière pièce rendue. Ou alors : dès la construction du tableau N_s, mémoriser également cette dernière pièce.

Mémoriser la dernière pièce

`dernierePiece[s]` vaut 1/7/23

sous forme là aussi de liste ou de dictionnaire. Par exemple pour une somme de 28 : `dernierePiece` vaut

[-1, 1, 1, 1, 1, 1, 1, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 23, 23, 23, 23, 23, 7]

Bien saisir le contenu de cette variable

- `dernierePiece[s]` n'est un choix pertinent que pour le cas s
- ce n'est qu'un des éléments de la solution ("dernière pièce")

L'utiliser pour trouver une solution optimale

d'abord sur l'exemple (une fois la dernière pièce traitée, que faire ?)
puis en général

Code pour remplir `dernierePiece`

```
def MonnaieRendue(somme):
    NbPiecesSuccessifs=[0]
    dernierePiece=[-1]
    for s in range(1,somme+1):
        N=NbPiecesSuccessifs[s-1]
        p=1
        if s>=7 and NbPiecesSuccessifs[s-7]<N:
            N=NbPiecesSuccessifs[s-7]
            p=7
        if s>=23 and NbPiecesSuccessifs[s-23]<N:
            N=NbPiecesSuccessifs[s-23]
            p=23
        NbPiecesSuccessifs.append(N+1)
        dernierePiece.append(p)
    return decodeOptimum(somme,dernierePiece)
```

Code d'exploitation

et voici le code pour l'exploiter : on décide de construire une liste de pièces à rendre.

```
def decodeOptimum(somme,piece):
    s=somme
    aRendre=[]
    while s>0:
        valeur=piece[s]
        aRendre.append(valeur)
        s=s-valeur
    return aRendre
```