Programmation dynamique

Résumé

Calculer le quarantième terme de la suite de Fibonacci par la définition récursive prend plusieurs secondes : la fonction recalcule des millions de fois les mêmes valeurs. La **programmation dynamique** corrige ce gaspillage. Lorsqu'un problème se ramène à des sous-problèmes qui se **chevauchent**, on calcule chaque sous-problème une seule fois et on mémorise son résultat, soit à la volée (**mémoïsation**), soit en remplissant un tableau du plus petit cas vers le plus grand (version **ascendante**). Là où un algorithme glouton fait un choix local sans jamais revenir en arrière, la programmation dynamique examine tous les choix possibles à partir des résultats déjà mémorisés : elle garantit l'optimum, par exemple pour rendre la monnaie avec le moins de pièces ou remplir un sac à dos.

Prérequis : La récursivité, Les dictionnaires, Tableaux à deux dimensions.

① Mémo

Définition 1 (Principe)

Sous-problèmes qui se chevauchent Le même sous-problème est rencontré plusieurs fois (`fib(3)` est demandé par `fib(5)` et par `fib(4)`). La récursion naïve le recalcule à chaque fois.

Mémoïsation (descendante) On garde la fonction récursive, mais on stocke chaque résultat dans un dictionnaire ; avant de calculer, on regarde s'il est déjà connu.

Version ascendante On remplit un tableau des plus petits cas vers le plus grand, sans récursion : chaque case se déduit de cases déjà remplies.

Comparaison Diviser pour régner : sous-problèmes *indépendants*. Glouton : un choix local à chaque étape, jamais remis en cause, optimum non garanti. Programmation dynamique : tous les choix examinés à partir des sous-résultats, optimum garanti.

Méthode en cinq étapes

1. Définir le **sous-problème** : que représente `T[i]` (ou `T[i][c]`) ?
2. Écrire la **relation de récurrence** : `T[i]` en fonction de cases plus petites.
3. Fixer les **cas de base** : `T[0]`, `T[1]`…
4. Choisir l'**ordre de remplissage** : toute case utilisée doit déjà être calculée.
5. Lire le **résultat** dans la dernière case, et au besoin **reconstruire** la solution (quels choix ont conduit à l'optimum ?).

Trois problèmes classiques

Fibonacci $F(n) = F(n-1) + F(n-2)$, $F(0) = 0$, $F(1) = 1$. Naïf : $O(2^n)$ appels. Dynamique : $O(n)$.

Rendu de monnaie $N(s)$, nombre minimal de pièces pour la somme s : $N(0) = 0$ et $N(s) = 1 + \min N(s-p)$ sur les pièces $p \leqslant s$. Le glouton (plus grosse pièce d'abord) échoue avec les pièces $\{1, 3, 4\}$ pour $s = 6$.

Sac à dos $T[i][c]$, valeur maximale avec les i premiers objets et une capacité c : $T[i][c] = \max\left(T[i-1][c],\ v_i + T[i-1][c-w_i]\right)$ si $w_i \leqslant c$, sinon $T[i][c] = T[i-1][c]$.

Pièges fréquents

- Utiliser un dictionnaire comme paramètre par défaut (`memo={}`) : il est partagé entre tous les appels de la fonction, y compris ceux qui n'ont rien à voir.
- Oublier un cas de base : `T[1]` manque et la relation lit une case inexistante.
- Remplir le tableau dans le mauvais ordre, en utilisant une case pas encore calculée.
- Confondre la valeur optimale (un nombre) et la solution optimale (la liste des choix) : la reconstruction demande un second parcours du tableau.
- Prendre le minimum d'un ensemble vide : si aucune pièce n'est utilisable, $N(s)$ doit valoir « infini » (`float('inf')`).

Erreurs classiques

Code erroné	Code correct	Explication

`def fib(n, memo={}):`	`def fib(n, memo=None):`
	`if memo is None:`
	`memo = {}`
		Le dictionnaire par défaut est créé une seule fois et partagé entre tous les appels : les résultats d'un calcul polluent le suivant.

`T = [0]` puis `T[1] = 1`	`T = [0, 1]`	La liste n'a qu'une case : `T[1] = 1` provoque un `IndexError`. On construit les cas de base avant la boucle.
`for i in range(n, 1, -1):` `T[i] = T[i-1] + T[i-2]`	`for i in range(2, n + 1):`	En descendant, `T[i-1]` n'est pas encore calculé : il faut remplir des petits indices vers les grands.
`N[s] = 1 + min(N[s - p] for p in pieces)`	`... for p in pieces if p <= s)`	Sans le filtre, `s - p` est négatif et `N[s - p]` lit la fin du tableau (indice négatif) : résultat faux sans message d'erreur.

② Exemples

Programme 1 · Fibonacci naïf : les appels explosent

```
appels = 0

def fib(n):
    global appels
    appels += 1
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)

print(fib(30), appels)   # 832040 2692537 : plus de 2,6 millions d'appels
```

Programme 2 · Mémoïsation (version descendante)

```
def fib_memo(n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)
    return memo[n]

print(fib_memo(30))    # 832040, chaque fib(k) calculé une seule fois
print(fib_memo(200))   # instantané
```

Programme 3 · Version ascendante : un tableau

```
def fib_tab(n):
    if n <= 1:
        return n
    T = [0, 1]                        # cas de base
    for i in range(2, n + 1):         # des petits indices vers les grands
        T.append(T[i - 1] + T[i - 2])
    return T[n]

print(fib_tab(30))   # 832040
```

Programme 4 · Rendu de monnaie : nombre minimal de pièces

```python
def nb_pieces(s, pieces):
    """Nombre minimal de pièces pour rendre la somme s (pièces disponibles en quantité illimitée)."""
    N = [0] + [float('inf')] * s          # N[0] = 0, les autres inconnus
    for montant in range(1, s + 1):
        for p in pieces:
            if p <= montant and N[montant - p] + 1 < N[montant]:
                N[montant] = N[montant - p] + 1
    return N[s]

print(nb_pieces(6, [1, 3, 4]))   # 2 (3 + 3), alors que le glouton rend 4 + 1 + 1
```

Tableau $\mathrm{N}(s)$ **pour les pièces** $\{1, 3, 4\}$:

s	**0**	**1**	**2**	**3**	**4**	**5**	**6**
$\mathrm{N}(s)$	0	1	2	1	1	2	2
pièce retenue		1	1	3	4	1 ou 4	3

Pour $s = 6$: $\mathrm{N}(6) = 1 + \min\big(\mathrm{N}(5),\ \mathrm{N}(3),\ \mathrm{N}(2)\big) = 1 + \min(2, 1, 2) = 2$, obtenu avec la pièce 3 (puis $\mathrm{N}(3) = 1$ avec une seconde pièce 3).

Programme 5 · Sac à dos : le tableau des valeurs

```python
def sac_a_dos(poids, valeurs, capacite):
    """Valeur maximale transportable (chaque objet pris au plus une fois).
    """
    n = len(poids)
    T = [[0] * (capacite + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):                    # objets 1 à n
        w, v = poids[i - 1], valeurs[i - 1]
        for c in range(capacite + 1):
            T[i][c] = T[i - 1][c]                # sans l'objet i
            if w <= c and v + T[i - 1][c - w] > T[i][c]:
                T[i][c] = v + T[i - 1][c - w]    # avec l'objet i
    return T

T = sac_a_dos([2, 3, 4], [3, 4, 5], 6)
print(T[3][6])   # 8
```

Tableau $\mathrm{T}[i][c]$ **pour les objets** $\mathrm{A}(2, 3)$, $\mathrm{B}(3, 4)$, $\mathrm{C}(4, 5)$ (poids, valeur) **et la capacité** 6 :

c	**0**	**1**	**2**	**3**	**4**	**5**	**6**
aucun objet	0	0	0	0	0	0	0
A	0	0	3	3	3	3	3
A, B	0	0	3	4	4	7	7
A, B, C	0	0	3	4	5	7	**8**

Reconstruction : T[3][6] = 8 ≠ T[2][6] = 7, donc C est pris et il reste la capacité 6 − 4 = 2 ; T[2][2] = 3 = T[1][2], donc B n'est pas pris ; T[1][2] = 3 ≠ T[0][2] = 0, donc A est pris. Solution optimale : {A, C}, valeur 8, poids 6.

③ Exercices

Exercice 1 *Compter les appels*

1. Dessiner l'arbre des appels de `fib(5)` (version naïve). Combien d'appels au total ? Combien de fois `fib(2)` est-il calculé ?
2. Avec la mémoïsation, combien de valeurs sont réellement calculées pour `fib_memo(5)` ?
3. Écrire `fib_deux(n)` qui calcule F(n) de façon ascendante en ne gardant que les deux derniers termes, sans tableau.

Exercice 2 *L'escalier* Pour monter un escalier de n marches, on avance à chaque pas d'une ou de deux marches. On note E(n) le nombre de façons de monter n marches, avec E(0) = 1 (ne rien faire) et E(1) = 1.

1. Justifier que E(n) = E($n-1$) + E($n-2$) pour $n \geqslant 2$.
2. Compléter le tableau des valeurs de E(n) pour n de 0 à 10.
3. Écrire `escalier(n)` en version ascendante.
4. On autorise aussi des pas de trois marches. Donner la nouvelle relation et calculer le nombre de façons de monter 10 marches.

Exercice 3 *Rendu de monnaie* Les pièces disponibles sont {1, 3, 4}, en quantité illimitée.

1. Compléter le tableau de N(s) pour s de 0 à 8.
2. Que rend l'algorithme glouton (plus grosse pièce d'abord) pour $s = 6$ et pour $s = 8$? Est-il optimal dans chaque cas ?
3. Écrire `nb_pieces(s, pieces)` en version ascendante.
4. Modifier la fonction pour qu'elle renvoie aussi la liste des pièces utilisées, en mémorisant pour chaque montant la pièce retenue.

Exercice 4 *Le sac à dos* Quatre objets sont disponibles, donnés sous la forme (poids, valeur) : $(1,1)$, $(3,4)$, $(4,5)$ et $(5,7)$. Le sac supporte un poids total de 7.

1. Appliquer l'algorithme glouton qui prend les objets par rapport valeur sur poids décroissant. Quelle valeur obtient-on ? Est-ce l'optimum ?
2. Remplir le tableau $T[i][c]$ pour i de 0 à 4 et c de 0 à 7.
3. Écrire `sac_a_dos(poids, valeurs, capacite)` qui renvoie la valeur maximale.
4. Reconstruire, à partir du tableau, la liste des objets choisis.

Solutions des exercices

Corrigé de l'exercice 1.

1. `fib(5)` appelle `fib(4)` et `fib(3)` ; `fib(4)` appelle `fib(3)` et `fib(2)` ; et ainsi de suite jusqu'aux feuilles `fib(1)` et `fib(0)`. Total : 15 appels ($1+2+4+6+2$ selon les niveaux). `fib(2)` est calculé 3 fois, `fib(3)` 2 fois.
2. Six valeurs seulement, F(0) à F(5) : chaque autre demande est servie par le dictionnaire.
3.

```python
def fib_deux(n):
    a, b = 0, 1          # F(0), F(1)
    for _ in range(n):
        a, b = b, a + b  # on avance d'un rang
    return a

print(fib_deux(30))   # 832040
```

Complexité $O(n)$ en temps et $O(1)$ en mémoire.

Corrigé de l'exercice 2.

1. Le dernier pas est soit une marche (il restait $n-1$ marches à monter avant, $E(n-1)$ façons), soit deux marches ($E(n-2)$ façons). Les deux cas sont disjoints et couvrent toutes les possibilités.
2.

n	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$E(n)$	1	1	2	3	5	8	13	21	34	55	89

Ce sont les nombres de Fibonacci décalés d'un rang.

3.

```python
def escalier(n):
    E = [1, 1]
    for i in range(2, n + 1):
        E.append(E[i - 1] + E[i - 2])
    return E[n]

print(escalier(10))   # 89
```

4. $E_3(n) = E_3(n-1) + E_3(n-2) + E_3(n-3)$ avec $E_3(0) = 1$, $E_3(1) = 1$, $E_3(2) = 2$. Les valeurs suivantes sont $4, 7, 13, 24, 44, 81, 149, 274$: il y a 274 façons de monter 10 marches.

Corrigé de l'exercice 3.

1.

s	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$N(s)$	0	1	2	1	1	2	2	2	2

Par exemple $N(7) = 1 + \min\big(N(6), N(4), N(3)\big) = 1 + 1 = 2$ (pièces $3 + 4$) et $N(8) = 1 + \min\big(N(7), N(5), N(4)\big) = 2$ (pièces $4 + 4$).

2. Pour 6 : le glouton rend 4, puis 1, puis 1, soit 3 pièces, alors que $3 + 3$ n'en demande que 2 : non optimal. Pour 8 : le glouton rend $4 + 4$, soit 2 pièces : optimal. Le glouton est parfois optimal, parfois non ; la programmation dynamique l'est toujours.
3.

```python
def nb_pieces(s, pieces):
    N = [0] + [float('inf')] * s
    for montant in range(1, s + 1):
        for p in pieces:
            if p <= montant and N[montant - p] + 1 < N[montant]:
                N[montant] = N[montant - p] + 1
    return N[s]

print(nb_pieces(6, [1, 3, 4]), nb_pieces(8, [1, 3, 4]))   # 2 2
```

4.

```python
def rendu(s, pieces):
    N = [0] + [float('inf')] * s
    choix = [None] * (s + 1)                    # pièce retenue pour
    chaque montant
    for montant in range(1, s + 1):
        for p in pieces:
            if p <= montant and N[montant - p] + 1 < N[montant]:
                N[montant] = N[montant - p] + 1
                choix[montant] = p
    liste = []
    reste = s
    while reste > 0:                            # reconstruction
        liste.append(choix[reste])
        reste -= choix[reste]
    return N[s], liste

print(rendu(6, [1, 3, 4]))   # (2, [3, 3])
print(rendu(7, [1, 3, 4]))   # (2, [3, 4])
```

Corrigé de l'exercice 4.

1. Rapports valeur sur poids : 1, $\approx 1{,}33$, 1,25, 1,4. Le glouton prend $(5, 7)$ puis $(1, 1)$ (poids 6, valeur 8), et ne peut plus rien ajouter. Or $(3, 4) + (4, 5)$ pèse 7 et vaut 9 : le glouton n'est pas optimal.
2.

c	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
aucun	0	0	0	0	0	0	0	0
$(1,1)$	0	1	1	1	1	1	1	1
$+(3,4)$	0	1	1	4	5	5	5	5
$+(4,5)$	0	1	1	4	5	6	6	9
$+(5,7)$	0	1	1	4	5	7	8	**9**

3.

```python
def sac_a_dos(poids, valeurs, capacite):
    n = len(poids)
    T = [[0] * (capacite + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        w, v = poids[i - 1], valeurs[i - 1]
        for c in range(capacite + 1):
            T[i][c] = T[i - 1][c]
            if w <= c and v + T[i - 1][c - w] > T[i][c]:
                T[i][c] = v + T[i - 1][c - w]
    return T[n][capacite]

print(sac_a_dos([1, 3, 4, 5], [1, 4, 5, 7], 7))   # 9
```

4. On remonte le tableau depuis T[4][7] = 9 : T[4][7] = T[3][7], l'objet $(5,7)$ n'est pas pris ; T[3][7] = 9 ≠ T[2][7] = 5, l'objet $(4,5)$ est pris et il reste $c = 3$; T[2][3] = 4 ≠ T[1][3] = 1, l'objet $(3,4)$ est pris et il reste $c = 0$; T[1][0] = T[0][0], l'objet $(1,1)$ n'est pas pris. Objets choisis : $(3,4)$ et $(4,5)$, valeur 9, poids 7.

```python
def objets_choisis(poids, valeurs, capacite):
    n = len(poids)
    T = [[0] * (capacite + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        w, v = poids[i - 1], valeurs[i - 1]
        for c in range(capacite + 1):
            T[i][c] = T[i - 1][c]
            if w <= c and v + T[i - 1][c - w] > T[i][c]:
                T[i][c] = v + T[i - 1][c - w]
    choisis = []
    c = capacite
    for i in range(n, 0, -1):
        if T[i][c] != T[i - 1][c]:       # l'objet i a été pris
            choisis.append(i - 1)
            c -= poids[i - 1]
    return T[n][capacite], choisis

print(objets_choisis([1, 3, 4, 5], [1, 4, 5, 7], 7))   # (9, [2, 1])
```