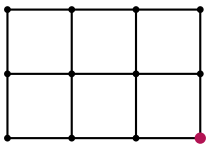
# *Chapitre 10*
# Programmation dynamique

Ce chapitre présente la **programmation dynamique** : une méthode de résolution de problèmes qui combine la **récursivité** (vue au chapitre précédent) et la **mémorisation** des résultats intermédiaires. L'idée générale est la suivante : pour résoudre un problème, on le décompose en **sous-problèmes** plus petits ; on résout chaque sous-problème *une seule fois*, on mémorise sa solution, et on la réutilise chaque fois que le même sous-problème réapparaît. Trois exemples classiques structurent ce chapitre : le comptage de chemins dans une grille, la suite de Fibonacci, puis le problème du rendu de monnaie.

## I. Compter des chemins dans une grille

Avant de parler de Fibonacci, commençons par un exemple simple : compter des chemins dans une grille.

**Exemple 10.1.**
On dispose de la grille $2 \times 3$ ci-dessous : deux rangées et trois colonnes de carreaux, donc trois rangées et quatre colonnes d'intersections.



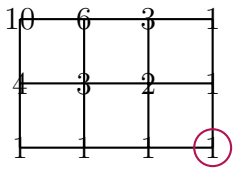
**Question :** combien de chemins mènent du coin inférieur droit (en olive) au coin supérieur gauche, en se déplaçant uniquement le long des **traits horizontaux vers la gauche** et le long des **traits verticaux vers le haut** ? ■

Une méthode efficace consiste à inscrire près de chaque intersection le nombre de chemins qui y mènent : ainsi, on n'a pas besoin de recompter à chaque fois que l'on veut repartir de ce coin.

**Exemple 10.2.**
Pour aller du coin inférieur droit à l'intersection située dans sa diagonale (une case vers la gauche puis une case vers le haut, ou l'inverse), on peut emprunter deux chemins. On inscrit alors « 2 » à l'intersection concernée. ■

La règle de remplissage est simple : pour atteindre une intersection, le dernier déplacement vient soit de l'intersection **à droite** (déplacement vers la gauche), soit de l'intersection **en dessous** (déplacement vers le haut). Le nombre de chemins qui mènent à une intersection est donc la **somme** du nombre de chemins de l'intersection à droite et de celui de l'intersection en dessous. Les intersections du bord inférieur et du bord droit ne peuvent être atteintes que d'une seule façon : elles reçoivent 1.

■ **Exemple 10.3.**
En continuant le raisonnement de bas en haut et de droite à gauche, on obtient la figure suivante : chaque intersection porte le nombre de chemins qui y mènent. Le coin de départ (inférieur droit) vaut 1, et le coin d'arrivée (supérieur gauche) vaut 10 : il y a donc **10 chemins** pour la grille $2 \times 3$. ■




■ **Exemple 10.4.**
À la main, on peut évaluer le nombre de chemins pour un carré de 5 par 5 : on remplit le tableau de $6 \times 6$ intersections avec la même règle (chaque case vaut la somme de la case à droite et de la case en dessous, les bords valent 1). On obtient 252 chemins. ■

■ **Exemple 10.5.**
Pour un tableau de 10 par 10, le même calcul donne 184 756 chemins. Le remplissage se fait très vite à la machine, alors qu'un comptage exhaustif des chemins serait irréaliste. ■

> **R** Les nombres obtenus sont les coefficients du **triangle de Pascal** : pour une grille de $n$ rangées et $m$ colonnes de carreaux, le nombre de chemins cherché est le coefficient binomial $\binom{n+m}{n}$. Cette formule n'est pas exigible en NSI, mais elle permet de vérifier ses résultats.

Le programme suivant implémente le remplissage du tableau : chaque case reçoit la somme de la case à droite et de la case en dessous, en parcourant le tableau de bas en haut et de droite à gauche.

```
def nb_chemins(nbligne, nbcol):
    tab = [[0] * (nbcol + 1) for _ in range(nbligne + 1)]
    for i in range(nbligne + 1):
        tab[i][nbcol] = 1
    for j in range(nbcol + 1):
        tab[nbligne][j] = 1
    for i in range(nbligne - 1, -1, -1):
        for j in range(nbcol - 1, -1, -1):
            tab[i][j] = tab[i + 1][j] + tab[i][j + 1]
    return tab
```

```
print(nb_chemins(2, 3)[0][0]) # 10 : grille 2 x 3
print(nb_chemins(5, 5)[0][0]) # 252 : grille 5 x 5
print(nb_chemins(10, 10)[0][0]) # 184756 : grille 10 x 10
```

Ce programme illustre déjà le principe de la programmation dynamique : chaque sous-problème (le nombre de chemins vers une intersection) est résolu **une seule fois**, son résultat est **mémorisé** dans le tableau, puis **réutilisé** pour calculer les cases suivantes.

# II. La suite de Fibonacci et la mémoïsation

## 1. La version récursive naïve

La suite de Fibonacci est définie par $F(0) = 1$, $F(1) = 1$ et, pour tout $n \geq 2$, $F(n) = F(n-1) + F(n-2)$. La programmation récursive donne une traduction directe de cette définition :

```
def fibo(n):
    if n <= 1:
        return 1
    else:
        return fibo(n - 1) + fibo(n - 2)
```

> **R** Si vous envisagez de faire le test avec des valeurs supérieures à 20, **sauvegardez avant** : la version récursive naïve devient extrêmement lente, car elle recalcule sans cesse les mêmes valeurs.

**▪ Exemple 10.6.**
Si on affiche l'arbre d'exécution de la fonction pour $n = 5$, on obtient le schéma suivant : chaque appel est représenté par un nœud, et les appels qui renvoient directement 1 sont les feuilles. ▪

```
fibo(5)
|-- fibo(4)
| |-- fibo(3)
| | |-- fibo(2)
| | | |-- fibo(1)
| | | |-- fibo(0)
| | |-- fibo(1)
| |-- fibo(2)
| |-- fibo(1)
| |-- fibo(0)
|-- fibo(3)
    |-- fibo(2)
    | |-- fibo(1)
    | |-- fibo(0)
    |-- fibo(1)
```

On constate que les appels à `fibo(0)` et `fibo(1)` sont très nombreux : **on recalcule plusieurs fois la même chose**. Le tableau suivant donne le nombre total d'appels effectués par la version naïve :

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 10 |
|---|---|---|---|---|---|---|---|
| Nombre d'appels | 1 | 1 | 3 | 5 | 9 | 15 | 177 |

Le nombre d'appels croît comme la suite de Fibonacci elle-même : il est **exponentiel**. Pour $n = 20$, il faut déjà 21 891 appels.

## 2. La mémoïsation

Le principe pour éviter ces recalculs consiste à **mémoriser temporairement les résultats déjà calculés**, dans une liste ou un dictionnaire. Cela s'appelle la **mémoïsation**.

**Définition 10.1.**
La **mémoïsation** consiste à stocker les résultats des sous-problèmes déjà résolus (dans une liste ou un dictionnaire) afin de ne pas les recalculer lors des appels suivants. On dit que l'on « mémoïse » les résultats.

■ **Exemple 10.7.**
On complète la fonction récursive : la liste `memoisation` contient les deux premiers termes connus, et chaque résultat calculé y est ajouté. Pour $n = 5$, la liste se remplit progressivement : `[1, 1]`, puis `[1, 1, 2]`, `[1, 1, 2, 3]`, `[1, 1, 2, 3, 5]` et enfin `[1, 1, 2, 3, 5, 8]` : le $6^e$ terme vaut 8. ■

```
memoisation = [1, 1] # les deux premiers termes valent 1

def fibonacci_dynamique(n, memoisation):
    if n < len(memoisation):
        return memoisation[n]
    resultat = fibonacci_dynamique(n - 1, memoisation) +
        fibonacci_dynamique(n - 2, memoisation)
    memoisation.append(resultat)
    return resultat
```

```
fibonacci_dynamique(5, memoisation) # 6e terme de Fibonacci : 8
fibonacci_dynamique(30, memoisation)
fibonacci_dynamique(400, memoisation)
```

Chaque terme n'est calculé qu'une seule fois : la fonction renvoie le 401$^{e}$ terme (un entier d'environ 84 chiffres) **instantanément**, alors que la version naïve ne terminerait pas.

**R** On ne recalcule pas plusieurs fois le même élément. On peut mesurer les temps d'exécution moyens des deux méthodes (version naïve et version mémoïsée) pour une même valeur de $n$ : la différence devient spectaculaire dès $n > 30$.

**R** À noter : la suite de Fibonacci ne sert qu'à **illustrer** le principe, car avec la forme itérative du calcul (une simple boucle qui garde les deux derniers termes), on ne répète pas plusieurs fois le même calcul : la mémoïsation n'est pas nécessaire en pratique pour Fibonacci.

**Définition 10.2.**
La **programmation dynamique** est une méthode algorithmique qui consiste à résoudre un problème en le décomposant en **sous-problèmes**, à résoudre chaque sous-problème **une seule fois**, puis à **mémoriser** ses solutions (généralement dans un tableau ou un dictionnaire) pour éviter de les recalculer. Les deux ingrédients sont : une **formule de récurrence** qui exprime la solution du problème à partir de celles des sous-problèmes, et une **mémoire** (tableau ou dictionnaire) qui stocke les solutions déjà calculées.

# III. Le rendu de monnaie

## 1. Le problème et l'algorithme glouton

Ce problème consiste à déterminer le nombre minimal de billets et de pièces nécessaires pour rendre une somme donnée. L'entrée est constituée de l'ensemble des valeurs des pièces $P$ et de la somme $s$ à rendre. La sortie est :

— soit le nombre minimal de pièces nécessaires (sortie 1) ;

— soit une liste de pièces, minimale, pour atteindre cette somme (sortie 2).

■ **Exemple 10.8.**
Le système des euros, avec des pièces à partir de 1 centime et des billets jusqu'à 500 €, correspond à l'entrée (les valeurs sont exprimées en **centimes**) :

$$P = [50\,000,\ 20\,000,\ 10\,000,\ 5\,000,\ 2\,000,\ 1\,000,\ 500,\ 200,\ 100,\ 50,\ 20,\ 10,\ 5,\ 2,\ 1]$$

On doit rendre 293,56 €, c'est-à-dire 29 356 centimes. L'**algorithme glouton** consiste à rendre à chaque étape la pièce de plus grande valeur inférieure ou égale à la somme restante :

$$29\,356 = 20\,000 + 5\,000 + 2\,000 + 2\,000 + 200 + 100 + 50 + 5 + 1$$

La sortie 1 est donc 9 pièces, et la sortie 2 est la liste `[20000, 5000, 2000, 2000, 200, 100, 50, 5, 1]`. ■

```
def rendu_glouton(liste_pieces, somme):
    ''' Renvoie la liste des pieces rendues par l'algorithme glouton.
        In : liste d'entiers triee par ordre decroissant, entier somme.
        Out : liste de pieces dont la somme vaut somme. '''
    rendu = []
    reste = somme
    for piece in liste_pieces:
        while piece <= reste:
            rendu.append(piece)
            reste = reste - piece
    return rendu
```

```
P = [50000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5,
    2, 1]
print(len(rendu_glouton(P, 29356))) # 9 pieces
```

■ **Exemple 10.9.**
L'algorithme glouton ne donne pourtant pas toujours une solution optimale. Avec $P_1 = [10, 6, 1]$ et $s = 12$, il rend `[10, 1, 1]` : 3 pièces. Or $12 = 6 + 6$ : 2 pièces suffisent. Le glouton a choisi la pièce de 10 en premier, et c'est ce choix qui le condamne. ■

**R** Pour toute la suite, on supposera que $1 \in P$. Cette hypothèse garantit qu'une **solution existe toujours** : avec des pièces de 1, on peut rendre n'importe quelle somme $s$ (il suffit d'en prendre $s$). Sans cette hypothèse, certaines sommes pourraient être impossibles à rendre.

## 2. Une formule de récurrence

On note $s$ la somme à rendre et $n_P(s)$ le nombre minimal de pièces à rendre pour la somme $s$.

**Propriété 10.1.**
Pour tout système de pièces $P$ contenant 1 et toute somme $s \geq 0$ :

$$n_P(s) = \begin{cases} 0 & \text{si } s = 0 \\ 1 + \min\limits_{p \in P,\ p \leq s} \big(n_P(s-p)\big) & \text{sinon} \end{cases}$$

Explication : pour rendre la somme $s$ avec une **première pièce** $p$, il reste $s - p$ à rendre, ce qui demande $n_P(s-p)$ pièces ; le nombre total est donc $1 + n_P(s-p)$. On choisit la pièce $p$ qui rend ce nombre minimal. Le cas $s = 0$ ne demande aucune pièce.

■ **Exemple 10.10.**

Appliquons la formule à $P = [10, 6, 1]$ et $s = 12$ :

$$n_P(12) = 1 + \min\big(n_P(2),\ n_P(6),\ n_P(11)\big) = 1 + \min(2,\ 1,\ 2) = 2$$

en effet $n_P(2) = 2$ $(1+1)$, $n_P(6) = 1$ (la pièce 6) et $n_P(11) = 2$ $(10+1)$. On retrouve bien la solution optimale : 2 pièces. ∎

```
def rendu_rec(P, s):
    ''' In : une liste P d'entiers et un entier s
        Out : nombre minimal de pieces de P dont la somme est s. '''
    if s == 0:
        return 0
    return 1 + min(rendu_rec(P, s - p) for p in P if p <= s)
```

```
print(rendu_rec([10, 6, 1], 12)) # 2
print(rendu_rec([10, 6, 1], 22)) # 3 (10 + 6 + 6)
```

**R** La fonction `rendu_rec` est-elle efficace ? **Non** : comme pour Fibonacci, l'arbre des appels récursifs explose et les mêmes calculs sont répétés des milliers de fois. Attention : éviter de prendre des valeurs de $s$ trop grandes avec cette version.

**R** **Pour aller plus loin (hors programme).** Pour visualiser les appels récursifs d'une fonction, on peut utiliser l'outil `recviz` : on associe à la fonction un **décorateur** `@recviz` (cette notion n'est pas au programme de Terminale, on se contente de l'utiliser), puis on appelle la méthode `callgraph()`. L'outil affiche chaque appel, ses arguments et sa valeur de retour. On peut l'installer avec `pip` et consulter sa documentation.

```
from recviz import recviz

@recviz
def fibo(n):
    if n <= 1:
        return 1
    return fibo(n - 1) + fibo(n - 2)
fibo(5)
```

## 3. La version récursive avec mémoïsation

On reprend le principe de la mémoïsation : un dictionnaire `dico_memoisation`, initialisé à $\{0 : 0\}$, associe peu à peu chaque somme $k$ $(0 \leq k \leq s)$ au nombre minimal $n_P(k)$ de pièces nécessaires. Le cas de base est : `somme` appartient à `dico_memoisation`.

```
def rendu_memo(liste_pieces, somme, dico_memoisation):
    if somme in dico_memoisation:
        return dico_memoisation[somme]
```

```
    dico_memoisation[somme] = 1 + min(
        rendu_memo(liste_pieces, somme - p, dico_memoisation)
        for p in liste_pieces if p <= somme)
    return dico_memoisation[somme]
```

■ **Exemple 10.11.**
Pour $P = [10, 6, 1]$ et $s = 12$, le dictionnaire se remplit au fil des appels : après le calcul, on obtient par exemple

$$\{0 : 0,\ 1 : 1,\ 2 : 2,\ 3 : 3,\ 4 : 4,\ 5 : 5,\ 6 : 1,\ 10 : 1,\ 11 : 2,\ 12 : 2\}$$

Chaque somme n'est calculée qu'une seule fois : le calcul de $n_P(12)$ est quasi instantané, là où `rendu_rec` explosait. ■

## 4. La version itérative : programmation dynamique

On peut améliorer l'algorithme précédent en évitant les appels récursifs. On utilise une liste $L$ de taille $s + 1$ que l'on initialise à $L[k] = k$ pour tout $0 \leq k \leq s$ (rendre $k$ avec $k$ pièces de 1), et que l'on met à jour de façon qu'à la fin, $L[k]$ soit le nombre minimal $n_P(k)$ de pièces dont la somme est $k$. On remplit la liste **de bas en haut** : chaque valeur ne dépend que de valeurs déjà calculées.

```
def rendu_dyn(P, s):
    ''' In : une liste P d'entiers et un entier s
        Out : nombre minimal de pieces de P dont la somme est s. '''
    L = [k for k in range(s + 1)] # initialisation : k pieces de 1
    for k in range(1, s + 1):
        for p in P:
            if p <= k:
                L[k] = min(L[k], 1 + L[k - p])
    return L[s]
```

```
print(rendu_dyn([10, 6, 1], 12)) # 2
P3 = [278, 100, 6, 4, 1]
print(rendu_dyn(P3, 100)) # 1 (la piece de 100)
```

■ **Exemple 10.12.**
Pour $P = [6, 4, 1]$ et $s = 8$, la liste $L$ se remplit de la façon suivante (seule la ligne du bas est la liste finale) : ■

| $k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| $L[k]$ | 0 | 1 | 2 | 3 | 1 | 2 | 1 | 2 | 2 |

On lit $L[8] = 2$ : il faut 2 pièces pour rendre 8 $(4 + 4)$, alors que l'algorithme glouton en utilisait 3 $(6 + 1 + 1)$.

> **R** **Coût de l'algorithme.** Pour chaque somme $k$ de 1 à $s$, on essaie chacune des $|P|$ pièces : le temps de calcul est en $O(s \times |P|)$. La mémoire utilisée est la liste $L$ de taille $s + 1$ : elle est en $O(s)$. La programmation dynamique **échange du temps contre de la mémoire** : on paye un coût en mémoire (le tableau des résultats) pour éviter des recalculs exponentiels.

## 5. Des systèmes de monnaie non canoniques

On dit qu'un système monétaire $P$ est **canonique** lorsque l'algorithme glouton donne toujours une solution optimale, quelle que soit la somme $s$.

■ **Exemple 10.13.**
Le système $P = [6, 4, 1]$ n'est pas canonique : pour $s = 8$, le glouton rend `[6, 1, 1]` (3 pièces) alors que la solution optimale est `[4, 4]` (2 pièces). ■

On peut chercher de tels contre-exemples à la main, ou automatiquement avec le module `random` : on tire des systèmes au hasard et on compare le nombre de pièces du glouton au nombre minimal donné par `rendu_dyn`. C'est l'objet d'un exercice de réflexion.

## 6. Reconstruire la liste des pièces

Le problème du rendu de monnaie admet deux sorties possibles : on sait calculer le nombre minimal de pièces, mais pas encore la **liste** des pièces. On peut adapter les algorithmes précédents pour qu'ils renvoient cette liste, mais il y a plus simple : à partir de la liste $L$ remplie par la version itérative, on peut reconstruire les pièces.

**Propriété 10.2.**
Soit $L$ la liste remplie par la version itérative. Pour reconstruire une liste minimale de pièces pour la somme $s$ : on part de $s$ ; d'après la façon dont $L$ est remplie, il existe une pièce $p$ telle que $L[s-p] = L[s] - 1$. On cherche cette pièce $p$, on la retient, et on continue avec $s - p$ jusqu'à arriver à $L[0] = 0$.

Il faut donc d'abord une fonction qui renvoie la liste $L$ complète : il suffit de modifier **une seule ligne** de `rendu_dyn` (renvoyer `L` au lieu de `L[s]`).

```
def rendu_dyn_pour_reconstruction(P, s):
    ''' In : une liste P d'entiers et un entier s
        Out : la liste L telle que pour tout entier 0 <= k <= s,
              L[k] = nombre minimal de pieces de P dont la somme est k. '''
    L = [k for k in range(s + 1)]
    for k in range(1, s + 1):
        for p in P:
            if p <= k:
                L[k] = min(L[k], 1 + L[k - p])
    return L
```

```
def rendu_reconstruction(P, s):
    ''' In : une liste P d'entiers et un entier s
        Out : liste de pieces de P, minimale, dont la somme vaut s. '''
    L = rendu_dyn_pour_reconstruction(P, s)
    pieces = []
    reste = s
    while reste > 0:
        for p in P:
            if p <= reste and L[reste - p] == L[reste] - 1:
                pieces.append(p)
                reste = reste - p
                break
    return pieces
```

■ **Exemple 10.14.**
Pour $P = [6, 4, 1]$ et $s = 8$, on a $L = [0, 1, 2, 3, 1, 2, 1, 2, 2]$. On part de 8 avec $L[8] = 2$ : la pièce 6 donne $L[2] = 2 \neq 1$, la pièce 4 donne $L[4] = 1 = L[8] - 1$ : on retient 4. Reste 4, avec $L[4] = 1$ : la pièce 4 donne $L[0] = 0 = L[4] - 1$ : on retient 4. Reste 0 : on s'arrête. La liste obtenue est `[4, 4]`. ■

## 7. Le cas des systèmes $[p^k, \ldots, p, 1]$

On cherche maintenant à savoir si le système monétaire $[p^k,\ p^{k-1},\ \ldots,\ p^2,\ p,\ 1]$ est canonique pour tout entier $p > 0$ et tout entier $k \geq 0$. Pour cela, on peut écrire une fonction qui teste tous les petits cas : trois boucles emboîtées (une sur $p$, une sur $k$, une sur $s$), et on renvoie `False` dès qu'un contre-exemple est trouvé.

```python
def sembleCanonique(p_max, k_max, s_max):
    for p in range(1, p_max + 1):
        for k in range(0, k_max + 1):
            P = [p ** i for i in range(k, -1, -1)]
            for s in range(0, s_max + 1):
                if len(rendu_glouton(P, s)) != rendu_dyn(P, s):
                    return False
    return True

print(sembleCanonique(100, 20, 20)) # True
```

■ **Exemple 10.15.**
La réponse est **oui** : pour tout $p > 0$ et tout $k \geq 0$, le système $[p^k, \ldots, p, 1]$ est canonique. En effet, chaque pièce est **p fois la précédente** : le glouton calcule exactement la décomposition de $s$ en **base** $p$, et on ne peut pas faire mieux (remplacer $p$ pièces de même valeur par une pièce $p$ fois plus grosse diminuerait toujours le nombre de pièces). Par exemple, avec $p = 3$ et $k = 4$, le système est $[81, 27, 9, 3, 1]$ et $100 = 81 + 9 + 9 + 1 = 10201_3$ : 4 pièces, et c'est le minimum. ■

# IV. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre : comptage de chemins, mémoïsation, Fibonacci, rendu de monnaie (glouton, récursif, mémoïsé, dynamique) et reconstruction. Pensez à bien lire les énoncés, à tester vos programmes, et à interpréter les messages d'erreur éventuels.

## Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : vocabulaire de la mémoïsation, lecture de code court, prédiction d'affichage, complétion. Chaque question doit pouvoir être traitée en moins de trente secondes.

**■ Exercice 10.1.**

Dire si les affirmations suivantes sont vraies ou fausses.

1. La mémoïsation consiste à recalculer plusieurs fois les mêmes résultats pour vérifier leur exactitude.
2. La mémoïsation consiste à mémoriser les résultats déjà calculés pour ne pas les recalculer.
3. L'algorithme glouton du rendu de monnaie donne toujours une solution optimale.
4. Si 1 appartient à l'ensemble des pièces, le problème du rendu de monnaie a toujours une solution.
5. La version récursive naïve de Fibonacci recalcule plusieurs fois les mêmes valeurs.

■

**■ Exercice 10.2.**

On exécute le programme suivant (la fonction `fibonacci_dynamique` est celle du cours) :

```
memoisation = [1, 1]
fibonacci_dynamique(4, memoisation)
print(memoisation)
```

Que contient la liste `memoisation` après cet appel ? ■

**■ Exercice 10.3.**

On considère la fonction récursive naïve :

```
def fibo(n):
    if n <= 1:
        return 1
```

```
    return fibo(n - 1) + fibo(n - 2)
```

1. Combien de fois `fibo(0)` est-elle appelée lors de l'exécution de `fibo(4)` ?
2. Combien d'appels à `fibo` cela fait-il au total ?

■

■ **Exercice 10.4.**
Rappel de Première : combien d'itérations la boucle suivante effectue-t-elle ?

```
for i in range(4, -1, -1):
    print(i)
```

■

■ **Exercice 10.5.**
Compléter la ligne manquante de la fonction `rendu_dyn` :

```
def rendu_dyn(P, s):
    L = [k for k in range(s + 1)]
    for k in range(1, s + 1):
        for p in P:
            if p <= k:
                L[k] = ____(L[k], 1 + L[k - p])
    return L[s]
```

Quelle fonction Python faut-il écrire à la place de `____` pour que la mise à jour soit correcte ?

■

### Exercices d'application

Les exercices d'application demandent d'écrire ou de compléter les fonctions du cours, puis de les tester sur des exemples.

■ **Exercice 10.6.**
Compléter la fonction `nb_chemins` pour qu'elle remplisse le tableau des chemins dans une grille (chaque case vaut la somme de la case à droite et de la case en dessous, les bords valent 1) :

```
def nb_chemins(nbligne, nbcol):
    tab = [[0] * (nbcol + 1) for _ in range(nbligne + 1)]
    for i in range(nbligne + 1):
        tab[i][nbcol] = 1
    for j in range(nbcol + 1):
        tab[nbligne][j] = 1
    for i in range(nbligne - 1, -1, -1):
        for j in range(nbcol - 1, -1, -1):
            ...
    return tab
```

1. Vérifier que `nb_chemins(3, 2)[0][0]` vaut bien 10.
2. Combien de chemins pour une grille de 5 par 5 ? Pour un tableau de 10 par 10 ?

■

■ **Exercice 10.7.**

Modifier la fonction `fibonacci_dynamique` du cours pour mémoïser les résultats dans un **dictionnaire** au lieu d'une liste. Le dictionnaire sera initialisé à $\{0 : 1,\ 1 : 1\}$.

```
dico = {0: 1, 1: 1}

def fibonacci_dico(n, dico):
    ...
```

Tester avec `fibonacci_dico(400, dico)`. ■

■ **Exercice 10.8.**

Écrire la fonction `rendu_glouton` qui renvoie la liste des pièces rendues par l'algorithme glouton (rendre la somme la plus grande inférieure ou égale à la somme qu'il reste à rendre), puis la tester.

```
def rendu_glouton(liste_pieces, somme):
    ''' In : une liste P d'entiers triee par ordre decroissant et un
        entier s
        Out : liste de pieces de P dont la somme vaut s,
              renvoyee par l'algorithme glouton '''
    pass
```

1. Vérifier que `rendu_glouton(P, 29356)` renvoie bien 9 pièces, avec $P$ le système des euros en centimes.
2. Vérifier que `rendu_glouton([100, 50, 20, 10, 5, 2, 1], 42)` renvoie

`[20, 20, 2]`.

■

■ **Exercice 10.9.**
À l'aide de la formule de récurrence du cours, écrire une fonction récursive `rendu_rec` qui prend en arguments un ensemble $P$ d'entiers et un entier $s$ et qui renvoie le nombre minimal $n_P(s)$ de pièces de $P$ dont la somme est $s$.

```
def rendu_rec(P, s):
    ''' In : une liste P d'entiers et un entier s
        Out : nombre minimal de pieces de P dont la somme est s '''
    pass
```

1. Tester avec $P_1 = [10, 6, 1]$ et $s = 12$, puis $s = 22$.
2. La fonction `rendu_rec` est-elle efficace ? Pourquoi ?

■

■ **Exercice 10.10.**
En reprenant le principe de mémoïsation, écrire une fonction récursive `rendu_memo` qui prend en arguments une liste `liste_pieces` d'entiers, un entier `somme` et un dictionnaire `dico_memoisation` initialisé à $\{0 : 0\}$, et qui renvoie le nombre minimal de pièces dont la somme est `somme`. Indication : le cas de base est `somme` $\in$ `dico_memoisation`.

```
def rendu_memo(liste_pieces, somme, dico_memoisation):
    ''' In : liste_pieces, somme, dico_memoisation initialise a {0: 0}
        Out : nombre minimal de pieces de liste_pieces dont la somme
              est somme '''
    pass
```

Tester avec $P_2 = [6, 4, 1]$ et vérifier que la liste des résultats pour $s$ de 0 à 10 vaut : `[0, 1, 2, 3, 1, 2, 1, 2, 2, 3, 2]`. ■

■ **Exercice 10.11.**
Proposer une fonction itérative `rendu_dyn` qui prend en arguments une liste `liste_pieces` d'entiers et un entier `somme` et qui renvoie le nombre minimal de pièces, en mettant à jour la liste `liste_memoisation` de taille $s + 1$.

```
def rendu_dyn(P, s):
    ''' In : une liste P d'entiers et un entier s
        Out : nombre minimal de pieces de P dont la somme est s '''
```

```
    pass
```

Vérifier que `[rendu_dyn([278, 100, 6, 4, 1], s) for s in range(20)]` vaut : `[0, 1, 2, 3, 1, 2, 1, 2, 2, 3, 2, 3, 2, 3, 3, 4, 3, 4, 3, 4]`. ■

### Exercices de réflexion

Les exercices de réflexion demandent de concevoir des programmes plus complets et d'analyser des situations limites.

■ **Exercice 10.12.**

**Un système non canonique.** Proposer un système de monnaie $P$ (de taille 4) contenant 1 qui ne soit pas canonique, c'est-à-dire tel qu'il existe une somme $s$ pour laquelle l'algorithme glouton ne donne pas la solution optimale.

1. Chercher un tel système à la main, en s'inspirant de $[10, 6, 1]$ et $[6, 4, 1]$. Donner le système $P$ et la somme $s$ qui conviennent, avec le rendu glouton et le rendu optimal.
2. On peut aussi chercher automatiquement : on utilise la fonction `randint` du module `random` et on compare `rendu_glouton` et `rendu_dyn`.

```
import random

def cherche_contre_exemple(taille, s_max, essais):
    for _ in range(essais):
        P = sorted(random.sample(range(2, 30), taille - 1), reverse=True
            ) + [1]
        for s in range(1, s_max + 1):
            if len(rendu_glouton(P, s)) != rendu_dyn(P, s):
                return P, s
    return None
```

■

■ **Exercice 10.13.**

**Reconstruire la liste des pièces.** On reprend le système $P = [10, 6, 4, 1]$.

1. Modifier légèrement (une seule ligne) la fonction `rendu_dyn` en une fonction `rendu_dyn_pour_reconstruction` qui renvoie la liste $L$ de taille $s + 1$ telle que pour tout $0 \leq k \leq s$, $L[k]$ soit le nombre minimal $n_P(k)$ de pièces dont la somme est $k$.
2. Écrire une fonction `rendu_reconstruction` qui prend en arguments un ensemble $P$ d'entiers et un entier $s$ et qui renvoie une liste de pièces de $P$, minimale, dont la somme est $s$ (on part de $s$ et on cherche à chaque étape une pièce $p$ telle que $L[s - p] = L[s] - 1$).
3. Tester avec `rendu_reconstruction([6, 4, 1], 8)` : on doit obtenir `[4,`

`4]`. Tester ensuite avec $P = [10, 6, 4, 1]$ et $s = 12$ : quelle liste obtient-on ? Vérifier que la somme vaut 12 et que le nombre de pièces est minimal.

■

■ **Exercice 10.14.**

**Les systèmes $[p^k, \ldots, p, 1]$ sont-ils canoniques ?** Écrire une fonction `sembleCanonique` qui prend en arguments des entiers `p_max`, `k_max` et `s_max` et qui renvoie `True` si l'algorithme glouton renvoie une solution optimale pour tout $1 \leq p \leq p\_max$, tout $0 \leq k \leq k\_max$ et tout $0 \leq s \leq s\_max$, et `False` sinon. Indication : trois boucles emboîtées suffisent, on construit le système $P = [p^k, \ldots, p, 1]$ avec une compréhension de liste, et on renvoie `False` dès que le glouton ne donne pas une solution optimale.

1. Tester avec `sembleCanonique(100, 20, 20)`. Que renvoie la fonction ?
2. Selon vous, le système monétaire $[p^k, \ldots, p, 1]$ est-il canonique pour tout entier $p > 0$ et tout entier $k \geq 0$ ? Justifier.

■

■ **Exercice 10.15.**

**Monter un escalier.** Un escalier a $n$ marches. À chaque pas, on monte soit 1 marche, soit 2 marches. On note $E(n)$ le nombre de façons différentes de monter l'escalier.

1. Expliquer pourquoi $E(0) = 1$, $E(1) = 1$ et, pour $n \geq 2$, $E(n) = E(n-1) + E(n-2)$.
2. Écrire une fonction `escaliers(n)` qui renvoie $E(n)$ en utilisant la programmation dynamique (version itérative, avec une liste remplie de bas en haut).
3. Vérifier que `escaliers(10)` vaut 89. Combien de façons pour 20 marches ?

```
def escaliers(n):
    E = [0] * (n + 1)
    ...
```

■