La programmation dynamique est une stratégie que l'on applique pour résoudre des problèmes. Elle repose largement sur le fait de construire un ou des tableaux stockant des résultats intermédiaires.

Elle s'applique particulièrement bien à la résolution de problèmes de dénombrements dans lesquels plusieurs paramètres sont présents et pour lesquels la programmation récursive donne de très mauvais résultats en terme de complexité.

1 Nombre de chemins

On considère une grille de taille $n \times m$ dans laquelle on peut se déplacer d'une case :

— vers le bas (i croissant) ;

— vers la droite (j croissant).

On cherche à trouver le nombre N de chemins allant de $(0,0)$ à $(n-1, m-1)$.

Définissons $N(i,j)$ le nombre de chemin allant de $(0,0)$ à (i,j) dans la grille. On a alors :

— pour tout j entre 0 et $m-1$

$$N(0,j) = 1$$

— pour tout i entre 0 et $n-1$

$$N(i,0) = 1$$

— pour tout couple (i,j) tel que $1 \leq i \leq n-1$ et $1 \leq j \leq m-1$

$$N(i,j) = N(i-1,j) + N(i,j-1)$$

Ces propriétés pourraient nous inviter à résoudre le problème de manière récursive, en écrivant la fonction suivante :

```
def nombre_chemins(i,j):
    if i==0 or j==0:
        return 1
    else:
        return nombre_chemins(i-1,j)+nombre_chemins(i,j-1)
```

Écrire et tester cette fonction sur des petites valeurs de n et m afin de vérifier les résultats puis sur des plus grandes valeurs. Que constate-t-on ?

2 Rendu de monnaie

On s'intéresse au problème de rendu de la monnaie. On se donne :

— une somme `s` à rendre ;

— un ensemble `pieces` de valeurs de pièces formant le système monétaire utilisé.

Les questions sont les suivantes :

— Quel est le nombre minimal de pièces à utiliser pour rendre la somme `s`

— Dans ce cas, quelles sont les pièces qui doivent être rendues ?

Une solution récursive donne facilement le programme suivant :

```
def rendu_monnaie(pieces, s):
    """renvoie le nombre minimal de pi ces pour faire
       la somme s avec le syst me pieces"""
    if s == 0:
        return 0
    r = s   # s = 1 + 1 + ... + 1 dans le pire des cas
    for p in pieces:
        if p <= s:
            r = min(r, 1 + rendu_monnaie(pieces, s - p))
    return r
```

Tester ce programme sur l'appel `rendu_monnaie([1,2,5], 10)` puis sur `rendu_monnaie([1,2], 100)`. Que constate-t-on ?

Ici, le problème provient de la redondance des calculs. Prenons dans le deuxième appel le cas du rendu de 98 pièces. Celui-ci est calculé récursivement sur l'appel du rendu de 100-2 mais aussi, plus tard, sur le rendu de 99-1. Cette répétitions dans les calculs se produit d'autant plus que la somme est petite (car il y a beaucoup de façon d'arriver à un faible rendu en partant de 100).

Pour résoudre ce problème : on procède comme suit. Plutôt que de procéder par appels récursifs de la fonction, on construit un tableau permettant de calculer et de garder en mémoire tous les résultats intermédiaires, ici en commençant par un rendu de 0.

```python
def rendu_monnaie(pieces, s):
    """renvoie le nombre minimal de pi ces pour faire
       la somme s avec le syst me pieces"""
    nb = [0] * (s + 1)
    for n in range(1, s + 1):
        nb[n] = n # n = 1 + 1 + ... + 1 dans le pire des cas
        for p in pieces:
            if p <= n:
                nb[n] = min(nb[n], 1 + nb[n - p])
    return nb[s]
```

Cette fois, l'algorithme proposé a une complexité quadratique, ce qui démontre une certaine efficacité.

Tester de nouveau programme sur l'appel `rendu_monnaie([1,2,5], 10)` puis sur `rendu_monnaie([1,2], 100)`. Que constate-t-on ?

Afin de résoudre totalement le problème, il est également possible de garder en mémoire la répartition des pièces rendues pour la solution considérée :

```python
def rendu_monnaie_solution(pieces, s):
    """renvoie une liste minimale de pi ces pour faire
       la somme s avec le syst me pieces"""
    nb = [0] * (s + 1)
    sol = [[]] * (s + 1)
    for n in range(1, s + 1):
        nb[n] = n
        sol[n] = [1] * n
        for p in pieces:
            if p <= n and 1 + nb[n - p] < nb[n]:
                nb[n] = 1 + nb[n - p]
                sol[n] = sol[n - p].copy()
                sol[n].append(p)
    return sol[s]
```

Tester le programme sur différents appels.

3 Alignement de séquences

Pour comparer des séquences d'ADN on cherche à résoudre le problème dit de l'alignement de séquences. Étant donnés deux mots, on essaie de mettre en correspondance le plus possible de lettres en conservant l'ordre des caractères mais en s'autorisant des trous `"-"`.

Par exemple, on peut aligner les mots `"COUCOU"` et `"COCO"` comme suit :

```
"COUCOU"
"CO-CO-"
```

Cet alignement semble bon mais on peut en imaginer de moins bons :

```
"COUCOU"
"-C-OCO"
```

Afin de distinguer les bons alignements des mauvais, on définit un **score** d'alignement :

— si deux caractères égaux sont alignés, on marque un point ;

— sinon (caractères `"-"` compris), on en perd un.

Ainsi l'alignement

```
"COUCOU"
"CO-CO-"
```

vaut 2 points alors que

```
"COUCOU"
"-C-OCO"
```

en vaut -6.

Résoudre le problème de l'alignement de séquence consiste à déterminer l'alignement (ou un alignement) de score maximal.

Réfléchissons d'un point de vu récursif sur notre exemple et ce à partir de la fin des mots : on a les caractères `"U"` et `"O"`. Trois possibilités s'offrent à nous :

1. ou bien on choisit d'aligner les deux caractères (quitte à perdre un point car ils sont ici différents) et dans ce cas, il restera à aligner `"COUCO"` et `"COC"` ;
2. ou bien on choisit d'aligner le `U` de `"COUCOU"` avec un trou `"-"` et dans ce cas, il restera à aligner `"COUCO"` et `"COCO"` ;
3. ou bien on choisit d'aligner le `O` de `"COCO"` avec un trou `"-"` et dans ce cas, il restera à aligner `"COUCOU"` et `"COC"`.

Remarque : Il n'est jamais intéressant d'aligner deux `"-"` car on n'avance pas dans le travail tout en perdant un point.

Plus précisément : si on veut aligner deux mots `s1` et `s2`, on commence par regarder récursivement les meilleurs score alignements pour :

1. `s1` et `s2` privés tous les deux de leur dernière lettre ;
2. `s1` privé de sa dernière lettre et s2 ;
3. `s1` et `s2` privé de sa dernière lettre.

En fonction de ses scores et des dernières lettre dont l'alignement peut faire gagner ou perdre des points, on peut alors trouver le meilleur score.

Quid des cas de base ? Puisque l'on diminue la taille des mots au fur et à mesure des appels, ils correspondent aux mots vides. Pour ceux-là, pas le choix : aligner un mot vide avec un autre de taille n consiste à mettre autant de trou que l'autre mot a de caractères, cela représente donc un alignement à $-n$ points.

```
"COUCOU"
"------"
```

Comme précédemment, on se doute que la programmation récursive risque d'impliquer des calculs redondants. Notre programme appliquera donc directement les méthodes de la programmation dynamique en construisant un tableau `sc` contenant le score maximal d'alignement de toutes les sous séquences des deux mots à aligner.

`sc[i][j] = score max pour aligner s1[0:i] et s2[0:j]`

Construisons à la main un tel tableau pour les petits mots `"BOB"` et `CUBE` en commençant par les cases représentant les cas de bases (première ligne et première colonne) :

		C	O	U	C	O	U
	0	-1	-2	-3	-4	-5	-6
C	-1	1	0	-1	-2	-3	-4
O	-2	0	2	1	0	-1	-2
C	-3	-1	1	1	2	1	0
O	-4	-2	0	0	1	3	2

Ces considérations amènent au code suivant :

```
def aligne(s1, s2):
    """le score du meilleur alignement de s1 et s2"""
    n1, n2 = len(s1), len(s2)
    sc = [[0] * (n2 + 1) for _ in range(n1 + 1)]
    # premi re ligne et premi re colonne
    for i in range(1, n1 + 1):
        sc[i][0] = -i
    for j in range(1, n2 + 1):
        sc[0][j] = -j
    # le reste
    for i in range(1, n1 + 1):
        for j in range(1, n2 + 1):
            s = max(-1 + sc[i - 1][j], -1 + sc[i][j - 1])
            if s1[i - 1] == s2[j - 1]:
                sc[i][j] = max(s,  1 + sc[i - 1][j - 1])
            else:
                sc[i][j] = max(s, -1 + sc[i - 1][j - 1])
    return sc[n1][n2]
```