

Programmation dynamique

Fiche de synthèse

① PRINCIPE

Définition 1 (Programmation dynamique) La **programmation dynamique** résout un problème à partir de sous-problèmes plus petits qui **se répètent** : chaque sous-problème est résolu une seule fois, et son résultat est conservé pour être réutilisé.

Avec diviser pour régner, les sous-problèmes sont **indépendants** : le tri fusion trie deux moitiés sans élément commun, et rien n'est à conserver. Pour la suite de Fibonacci, au contraire, le calcul de F_{40} demande F_{38} deux fois, F_{37} trois fois, et ainsi de suite.

② LA SUITE DE FIBONACCI

$F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$. La version récursive directe recalcule sans cesse les mêmes termes : `fibonacci(40)` prend plusieurs secondes.

Mémoïsation, ou approche **descendante** : on garde la fonction récursive et on range chaque terme calculé dans un tableau.

```
def fibo_memo(n, tableau):
    if n == 0 or n == 1:
        return n
    if tableau[n] == 0:
        a = fibo_memo(n - 1, tableau)
        b = fibo_memo(n - 2, tableau)
        tableau[n] = a + b
    return tableau[n]

print(fibo_memo(40, [0] * 41))
```

Approche **ascendante** : on remplit le tableau en partant des plus petits cas.

```
def fibo_dyn(n):
    if n == 0:
        return 0
    tableau = [0] * (n + 1)
    tableau[1] = 1
    for i in range(2, n + 1):
        tableau[i] = (tableau[i - 1]
                     + tableau[i - 2])
    return tableau[n]
```

Chaque terme est calculé une seule fois : le calcul de F_n demande un nombre d'opérations proportionnel à n .

③ LE PROBLÈME DU SAC À DOS

Chaque objet a une valeur et un poids ; le sac supporte au plus un poids w . On cherche les objets qui donnent la plus grande valeur totale sans dépasser w . La force brute essaie les 2^n choix possibles ; un algorithme glouton est rapide mais ne garantit pas le meilleur choix.

☞ **Sac à dos par programmation dynamique.** On note $T[i][p]$ la meilleure valeur obtenue avec les i premiers objets et un sac de capacité p . Pour le i -ième objet, de valeur v et de poids q :

- on le **laisse** : la valeur est $T[i - 1][p]$;
- on le **prend**, si $q \leq p$: la valeur est $v + T[i - 1][p - q]$.

$T[i][p]$ est la plus grande de ces deux valeurs. Chaque ligne du tableau se calcule à partir de la précédente.

Pour les objets (3, 2), (8, 10), (2, 2), (8, 1), (4, 6) et (6, 6), au format (valeur, poids), et $w = 10$:

$i \backslash p$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3	3
2	0	0	3	3	3	3	3	3	3	3	8
3	0	0	3	3	5	5	5	5	5	5	8
4	0	8	8	11	11	13	13	13	13	13	13
5	0	8	8	11	11	13	13	13	13	15	15
6	0	8	8	11	11	13	13	14	14	17	17

La meilleure valeur est 17. On retrouve les objets en remontant le tableau : quand une valeur diffère de celle de la ligne du dessus, l'objet de cette ligne est pris, et l'on retire son poids à la capacité. On obtient les objets 1, 4 et 6, de poids total 9 kg.

```
def sac_a_dos(objets, w):
    n = len(objets)
    T = [[0] * (w + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        v, q = objets[i - 1]
        for p in range(w + 1):
            T[i][p] = T[i - 1][p]
            if q <= p:
                avec = v + T[i - 1][p - q]
                T[i][p] = max(T[i][p], avec)
    return T[n][w]
```

④ DISTANCE D'ÉDITION

Définition 2 (Distance d'édition) La **distance d'édition** entre deux chaînes est le plus petit nombre de modifications, **substitution**, **insertion** ou **suppression** d'un caractère, qui transforme la première en la seconde.

La distance de "CHAT" à "CHATS" vaut 1 ; celle de "NICHE" à "CHIEN" vaut 4. On compare ainsi des séquences d'ADN.

On note $d(i, j)$ la distance entre les i premiers caractères de A et les j premiers de B ; c vaut 0 si $A[i - 1] = B[j - 1]$ et 1 sinon. Alors $d(0, j) = j$, $d(i, 0) = i$ et

$d(i, j) = \min(d(i - 1, j) + 1, d(i, j - 1) + 1, d(i - 1, j - 1) + c)$.

```
def distance_edition(a, b):
    n, m = len(a), len(b)
    t = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(n + 1):
        t[i][0] = i
    for j in range(m + 1):
        t[0][j] = j
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            c = 0 if a[i - 1] == b[j - 1] else 1
            t[i][j] = min(t[i - 1][j] + 1,
                          t[i][j - 1] + 1,
                          t[i - 1][j - 1] + c)
    return t[n][m]
```

La version récursive directe déclenche trois appels à chaque étape et devient exponentielle ; le tableau rempli ligne par ligne demande un temps et une mémoire en $O(n \times m)$.

📌 À retenir.

- La programmation dynamique s'applique quand les sous-problèmes se répètent ; diviser pour régner, quand ils sont indépendants.
- La mémorisation garde la récursivité et range les résultats ; l'approche ascendante remplit un tableau depuis les plus petits cas.
- Sac à dos : on compare, pour chaque objet et chaque capacité, le choix de le laisser et celui de le prendre.
- La distance d'édition se calcule dans un tableau en $O(n \times m)$.