Programmation dynamique

Algorithmique

① Activité 1 : la suite de Fibonacci

1.1 Récursivité

La suite de Fibonacci commence par les termes $F_0 = 0$ et $F_1 = 1$; ensuite, chaque terme est la somme des deux qui le précèdent : $F_n = F_{n-1} + F_{n-2}$ pour tout entier $n \geq 2$.

Ses premiers termes sont donc 0, 1, 1, 2, 3, 5, 8, 13, 21 et 34. On numérote les termes à partir de 0 : le terme numéro 7, noté F_7, vaut 13.

Exercice 1 *Fibonacci récursif naïf* Écrire une fonction récursive `fibo(n)` qui renvoie F_n, puis l'utiliser pour calculer F_{10}, F_{20}, F_{30} et F_{40}. Combien de temps faut-il pour obtenir `fibo(40)` ? D'où vient cette lenteur ?

1.2 Approche récursive par mémoïsation (top-down)

On garde la fonction récursive, mais on range chaque terme calculé dans un tableau : avant tout calcul, la fonction regarde si le terme demandé s'y trouve déjà. Cette technique s'appelle la **mémoïsation**.

```
def fibo_memo(n, tableau=None):
    if tableau is None:
        tableau = [None] * (n + 1)
    if tableau[n]:
        return tableau[n]
    if n == 0 or n == 1:
        return n
    tableau[n] = fibo_memo(n - 1, tableau) + fibo_memo(n - 2, tableau)
    return tableau[n]

print(fibo_memo(710))
```

Le résultat s'affiche aussitôt, même pour $n = 710$: chaque terme n'est calculé qu'une fois, le coût est en $O(n)$.

1.3 Approche ascendante (bottom-up)

On peut aussi se passer de la récursivité : on remplit le tableau dans l'ordre, du terme 0 au terme n, chaque case étant la somme des deux précédentes.

```
def fibo_dyn(n):
    if n == 0:
        return 0
    tableau = [0] * (n + 1)
    tableau[1] = 1 # deuxième terme de la suite de Fibonacci
```

```
    for i in range(2, n + 1):
        tableau[i] = tableau[i - 1] + tableau[i - 2]
    return tableau[n]

print(fibo_dyn(40))
print(fibo_dyn(710))
```

Le coût est encore en $O(n)$, et aucun appel récursif ne vient remplir la pile d'appels.

② Activité 2 : le sac à dos

2.1 Rappel du problème

Le problème du sac à dos a été rencontré en première. Chaque objet est décrit par un couple (valeur, poids) ; le sac supporte au plus un poids w. Il s'agit de choisir les objets à emporter pour que leur valeur totale soit la plus grande possible, sans dépasser w.

2.2 Résolution à l'aide d'un algorithme glouton

En première, deux méthodes ont été étudiées :

— la **force brute** essaie tous les choix d'objets possibles ; avec n objets, il y en a 2^n : un million pour 20 objets, déjà trop pour 60 ;

— les **algorithmes gloutons** trient les objets selon un critère puis les prennent dans cet ordre tant qu'ils tiennent dans le sac :

  — premier glouton : les objets les plus précieux d'abord ;

  — second glouton : les objets dont le rapport valeur/poids est le plus élevé d'abord.

Un glouton ne coûte que le tri des objets, soit $O(n \log_2 n)$ avec un tri efficace ; en revanche, rien ne garantit que son choix soit le meilleur.

2.3 Résolution par programmation dynamique

L'idée consiste à traiter les objets un par un, et à se servir des résultats obtenus avec les premiers objets pour trouver le meilleur choix quand on en ajoute un.

Notons (v_i, p_i) le i-ème objet. Supposons connue, pour chaque capacité de sac p comprise entre 0 et w, la meilleure valeur réalisable avec les $i-1$ premiers objets. Pour la capacité p, quand le i-ème objet devient disponible, deux possibilités se présentent :

— on **laisse** le i-ème objet : la meilleure valeur est celle obtenue avec les $i-1$ premiers objets et la capacité p ;

— on **prend** le i-ème objet, s'il pèse au plus p : il apporte la valeur v_i, et il reste la capacité $p - p_i$ pour les $i-1$ premiers objets.

On garde la meilleure des deux. Les résultats se rangent dans un tableau : une ligne par nombre d'objets disponibles (de 0 à n), une colonne par capacité (de 0 à w). Chaque ligne se calcule à partir de la précédente.

2.4 Exemple

On prend `objets = [(3, 2), (8, 10), (2, 2), (8, 1), (4, 6), (6, 6)]`, au format (valeur, poids), et un sac de capacité $w = 10$ kg.

objets \ poids	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 case en bas à droite donne la meilleure valeur : **17**. Pour retrouver les objets choisis, on remonte le tableau à partir de la colonne 9, la plus petite capacité où 17 apparaît dans la dernière ligne :

— ligne 6, capacité 9 : 17, alors que la ligne 5 donne 15 ; l'objet 6 (6 kg) est pris, et l'on poursuit avec la capacité $9 - 6 = 3$;

— ligne 5, capacité 3 : 11, comme à la ligne 4 ; l'objet 5 n'est pas pris ;

— ligne 4, capacité 3 : 11, alors que la ligne 3 donne 3 ; l'objet 4 (1 kg) est pris, il reste $3 - 1 = 2$ kg ;

— ligne 3, capacité 2 : 3, comme aux lignes 2 et 1, et la ligne 0 donne 0 : seul l'objet 1 (2 kg) est pris.

Le meilleur choix est donc formé des objets 1, 4 et 6, pour une valeur de 17 et un poids de 9 kg.

Exercice 2 *Sac à dos – tableau* Construire le tableau pour `objets = [(4, 3), (7, 4), (5, 2), (9, 5), (3, 1), (6, 3)]` et $w = 8$, puis retrouver les objets choisis.

③ Programmation en Python

Exercice 3 *Sac à dos en Python* Résoudre le problème du sac à dos dans le cas où `objets = [["A", 6, 5], ["B", 3, 2], ["C", 3, 2], ["D", 3, 2], ["E", 1, 1]]` (format : nom, valeur, poids) et $w = 6$:

1. à l'aide d'un algorithme glouton (version glouton 1 : on prend d'abord les objets de valeur maximale) ;
2. par programmation dynamique.

④ Programmation dynamique : synthèse

Dans les deux activités, la même démarche apparaît : on ramène le problème à des problèmes plus petits, on résout d'abord les plus petits, et l'on **conserve leurs résultats** pour construire ceux des plus grands. C'est le principe de la **programmation dynamique**.

La méthode « diviser pour régner » découpe elle aussi un problème en sous-problèmes, mais ceux-ci sont **indépendants** : le tri fusion trie deux moitiés qui n'ont aucun élément commun, et chaque moitié n'est triée qu'une fois.

Dans la suite de Fibonacci ou le sac à dos, au contraire, les sous-problèmes **se répètent** : le calcul de F_{40} demande F_{38} deux fois, F_{37} trois fois, et ainsi de suite. Un découpage récursif sans mémoire refait ces calculs ; la programmation dynamique les fait une seule fois et range chaque résultat dans un tableau.

Méthode	Caractéristique
Diviser pour régner	Sous-problèmes **indépendants**, pas de chevauchement
Programmation dynamique	Sous-problèmes **chevauchants**, stockage des résultats

Remarque. Deux conditions rendent la méthode efficace : le meilleur choix pour le problème se construit à partir des meilleurs choix pour des problèmes plus petits (**sous-structure optimale**), et ces problèmes plus petits reviennent plusieurs fois (**sous-problèmes chevauchants**).

⑤ Activité 3 : alignement de séquences

5.1 Présentation du problème

En bio-informatique, on compare deux séquences d'ADN pour savoir à quel point elles se ressemblent. Une mesure possible : le plus petit nombre de modifications qui transforme une chaîne de caractères en l'autre.

Chaque modification compte pour 1 et peut être :

— **substitution** : remplacer un caractère par un autre (coût 1) ;

— **insertion** : insérer un caractère (coût 1) ;

— **suppression** : supprimer un caractère (coût 1).

Le plus petit nombre de modifications nécessaires s'appelle la **distance d'édition** des deux chaînes (ou distance de Levenshtein).

Exemple. Pour transformer `"CHAT"` en `"CHATS"`, il suffit d'une insertion (ajouter le `S`). La distance est donc 1. Pour transformer `"NICHE"` en `"CHIEN"`, la distance est 4 : supprimer N, supprimer I, insérer I après CH, insérer N.

5.2 Relation de récurrence

Pour deux chaînes A et B, on note $d(i, j)$ la distance d'édition entre le début de A formé de ses i premiers caractères et le début de B formé de ses j premiers caractères.

Cas de base :

— si $i = 0$: $d(0, j) = j$ (il faut j insertions) ;

— si $j = 0$: $d(i, 0) = i$ (il faut i suppressions).

Cas général :

$$d(i,j) = \min \begin{cases} d(i-1,j)+1 & \text{(suppression)} \\ d(i,j-1)+1 & \text{(insertion)} \\ d(i-1,j-1)+c & \text{(substitution si } c=1\text{, rien si } c=0) \end{cases}$$

où $c = 0$ si $\mathrm{A}[i-1] = \mathrm{B}[j-1]$, et $c = 1$ sinon.

5.3 Approche récursive naïve

```python
def distance_recursive(a, b):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    cout = 0 if a[-1] == b[-1] else 1
    return min(
        distance_recursive(a[:-1], b) + 1,
        distance_recursive(a, b[:-1]) + 1,
        distance_recursive(a[:-1], b[:-1]) + cout
    )

print(distance_recursive("CHAT", "CHATS"))
print(distance_recursive("NICHE", "CHIEN"))
```

Cette fonction donne le bon résultat, mais chaque appel en déclenche trois autres : les mêmes débuts de chaînes sont comparés un très grand nombre de fois, comme les termes de Fibonacci dans `fibo`.

5.4 Résolution par programmation dynamique

Si les chaînes ont pour longueurs n et m, on remplit un tableau de $n+1$ lignes et $m+1$ colonnes, ligne par ligne : la case de la ligne i et de la colonne j reçoit $d(i, j)$, calculée à partir de trois cases déjà remplies.

```python
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):
            cout = 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] + cout
            )
    return t[n][m]

print(distance_edition("CHAT", "CHATS"))
print(distance_edition("NICHE", "CHIEN"))
print(distance_edition("ALGORITHME", "ALTRUISME"))
```

Résultats : 1, 4, 5.

Chaque case demande un nombre constant d'opérations : le temps de calcul et la place occupée sont en $O(n \times m)$, alors que la version récursive est exponentielle.

5.5 Visualisation du tableau

Pour A = "CHAT" et B = "CHATS" :

		C	H	A	T	S
	0	1	2	3	4	5
C	1	0	1	2	3	4
H	2	1	0	1	2	3
A	3	2	1	0	1	2
T	4	3	2	1	0	1

La case en bas à droite contient la distance cherchée, 1. En remontant depuis cette case vers la case voisine qui a fourni le minimum, on retrouve la suite des modifications.

Exercice 4 *Construction du tableau*

1. Construire à la main le tableau pour les chaînes `"BAIN"` et `"BRIN"`.
2. Quelle est la distance d'édition ?
3. Remonter le chemin optimal et lister les opérations effectuées.

Exercice 5 *Application en bio-informatique* En bio-informatique, les séquences ADN sont des chaînes composées des lettres A, C, G, T. On considère les deux séquences suivantes :

$$S_1 = \texttt{"ACGTAC"} \quad \text{et} \quad S_2 = \texttt{"AGTC"}$$

1. Calculer la distance d'édition entre S_1 et S_2 à l'aide de la fonction `distance_edition`.
2. Construire le tableau correspondant.
3. Proposer un alignement optimal en utilisant le symbole `-` pour les insertions et suppressions.

Exercice 6 *Modification de la fonction* Modifier la fonction `distance_edition` pour qu'elle renvoie non seulement la distance, mais aussi la liste des opérations effectuées (substitution, insertion, suppression ou identité).