Programmation dynamique : activités de découverte

Éviter les recalculs, construire des solutions par étapes

Remarque 1 (Avant de commencer). Ces activités ont pour but de vous faire *découvrir* les idées-clés de la programmation dynamique avant l'étude du cours. Vous y rencontrerez trois problèmes très différents (Fibonacci, le sac à dos, la distance d'édition) qui partagent une même structure : un problème se décompose en sous-problèmes qui se répètent. On va apprendre à ne jamais calculer deux fois la même chose.

① Fibonacci et l'explosion combinatoire

On rappelle la suite de Fibonacci, définie par $F_0 = 0$, $F_1 = 1$ et pour tout entier $n \geqslant 2$, $F_n = F_{n-1} + F_{n-2}$. Les premiers termes sont : $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

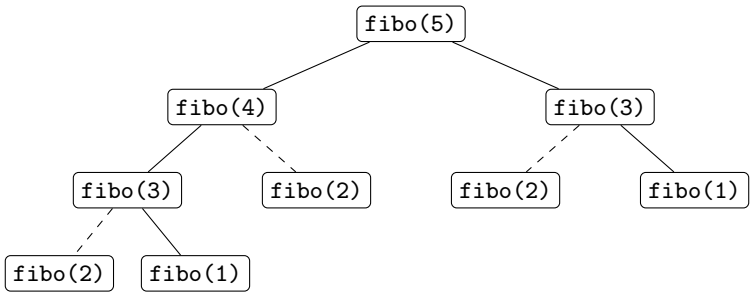
Une traduction directe en Python de cette définition donne la fonction récursive suivante :

```
def fibo(n):
    if n == 0 or n == 1:
        return n
    return fibo(n - 1) + fibo(n - 2)
```

Cette fonction est correcte mais présente un défaut majeur que nous allons mettre en évidence.

Exercice 1 *L'arbre d'appels de `fibo(5)`* On appelle « arbre d'appels » d'un appel récursif l'arbre dont chaque nœud est étiqueté par un appel de la fonction, et dont les fils d'un nœud sont les appels récursifs déclenchés par ce nœud.

Voici le début de l'arbre d'appels de `fibo(5)` :



Les branches en pointillés signalent que l'arbre n'est pas complet à cet endroit : il faut prolonger chaque `fibo(2)` en ses deux fils `fibo(1)` et `fibo(0)` (qui sont, eux, des feuilles).

1. Recopier l'arbre d'appels complet de `fibo(5)` sur votre cahier.
2. Combien de feuilles l'arbre contient-il en tout ? Combien sont étiquetées `fibo(0)` ? `fibo(1)` ?
3. En déduire le nombre total d'additions effectuées lors du calcul de `fibo(5)`.

Exercice 2 *Les recalculs sont exponentiels* Notons $C(n)$ le nombre d'appels de fonction effectués lors du calcul de `fibo(n)` (en comptant l'appel initial).

1. Justifier que pour tout entier $n \geqslant 2$, on a $C(n) = 1 + C(n-1) + C(n-2)$, avec $C(0) = C(1) = 1$.
2. Calculer $C(2)$, $C(3)$, $C(4)$, $C(5)$, $C(6)$ et $C(10)$.
3. Comparer $C(n)$ à F_n. Que remarque-t-on ?
4. Pourquoi cette fonction est-elle de fait inutilisable pour calculer `fibo(50)` ?

② Mémoïser pour ne jamais recalculer

L'idée de la **mémoïsation** est simple : chaque fois que l'on calcule la valeur d'un sous-problème, on l'*enregistre* dans un tableau (ou un dictionnaire). La prochaine fois que le même sous-problème se présente, on lit directement la valeur déjà calculée.

Exercice 3 *`fibo(5)` à la main avec mémoïsation* Reprenons le calcul de `fibo(5)` mais en remplissant cette fois un tableau T à 6 cases, initialisé à `None`. Dès qu'une valeur F_k est calculée, on la range dans $T[k]$. Si $T[k]$ est déjà renseigné, on le lit sans le recalculer. La fonction récursive avec mémoïsation s'écrit :

```
def fibo_memo(n, T):
    if T[n] is not None:    # déjà calculé : on lit le cache
        return T[n]
    if n == 0 or n == 1:    # cas de base
        T[n] = n
        return n
    T[n] = fibo_memo(n - 1, T) + fibo_memo(n - 2, T)
    return T[n]
```

On part d'un tableau `T` de 6 cases, toutes initialement à `None` :

indice k	0	1	2	3	4	5
$T[k]$						

1. Recopier le tableau, puis simuler pas à pas l'appel `fibo_memo(5, T)`. Détailler l'ordre dans lequel les cases $T[k]$ sont remplies.
2. Combien d'additions effectue-t-on en tout avec cette version ? Comparer avec l'exercice précédent.
3. Pour passer de `fibo(5)` à `fibo(50)`, le nombre d'additions est-il désormais exponentiel ?

Exercice 4 *De la mémoïsation au « bottom-up »* On peut faire le même calcul *sans récursivité*, en remplissant directement le tableau T de gauche à droite, case par case : on connaît $T[0]$ et $T[1]$, et pour tout $k \geqslant 2$ on écrit $T[k] = T[k-1] + T[k-2]$. Cette approche s'appelle l'approche **ascendante** (en anglais : *bottom-up*).

1. Recopier et compléter le tableau suivant pour calculer F_{10} :

k	0	1	2	3	4	5	6	7	8	9	10
$T[k]$	0	1									

2. Écrire en langage algorithmique ou en Python une fonction `fibo_dyn(n)` qui renvoie F_n en suivant cette méthode.
3. Comparer les trois versions (`fibo`, `fibo_memo`, `fibo_dyn`) du point de vue du nombre d'additions et de l'utilisation de la mémoire.

Remarque 2 (Les deux visages de la programmation dynamique). Les deux approches (*top-down* avec mémoïsation, *bottom-up* avec un tableau rempli dans l'ordre) sont strictement équivalentes en nombre de sous-problèmes résolus. L'approche *bottom-up* est souvent plus rapide (pas d'appels de fonction, pas de pile de récursivité) mais demande d'avoir identifié à l'avance l'ordre dans lequel les sous-problèmes doivent être résolus.

③ Le sac à dos par programmation dynamique

On dispose de n objets, chacun caractérisé par une valeur v_i et un poids p_i. On veut remplir un sac à dos dont la capacité maximale est un poids W, en maximisant la somme des valeurs des objets emportés. Chaque objet peut être pris au plus une fois (sac à dos 0/1).

Prenons W = 8 kg et les quatre objets suivants :

objet i	1	2	3	4
valeur v_i	15	11	10	8
poids p_i	6	4	4	3

Exercice 5 *L'approche gloutonne ne suffit pas* On rappelle deux stratégies gloutonnes vues en Première :

— *glouton valeurs* : on prend les objets par valeur décroissante, tant que le poids le permet ;

— *glouton ratio* : on prend les objets par rapport v_i/p_i décroissant, tant que le poids le permet.

1. Appliquer les deux stratégies gloutonnes sur l'exemple ci-dessus. Quelle valeur totale obtient-on dans chaque cas ?
2. Trouver « à la main » une combinaison d'objets qui donne une valeur totale strictement supérieure à celles obtenues par les deux stratégies gloutonnes. Que peut-on en conclure sur la fiabilité des stratégies gloutonnes pour ce problème ?

Exercice 6 *Remplir le tableau de programmation dynamique* On note $\mathrm{T}[i][p]$ la valeur maximale que l'on peut obtenir avec les i premiers objets et un poids maximal p. On convient que $\mathrm{T}[0][p] = 0$ pour tout p (aucun objet disponible) et $\mathrm{T}[i][0] = 0$ pour tout i (aucune capacité disponible).

La relation de récurrence est la suivante, pour $i \geqslant 1$ et $p \geqslant 0$:

$$\mathrm{T}[i][p] = \begin{cases} \mathrm{T}[i-1][p] & \text{si } p_i > p \text{ (l'objet ne rentre pas) ;} \\ \max\big(\mathrm{T}[i-1][p]\,,\ \mathrm{T}[i-1][p-p_i] + v_i\big) & \text{sinon.} \end{cases}$$

Autrement dit, pour chaque nouvel objet i et chaque capacité p, on compare le fait de *ne pas prendre* l'objet i (on garde alors le résultat de la ligne précédente) au fait de *prendre* l'objet i (si la place le permet).

1. Recopier le tableau suivant et le remplir ligne par ligne, en explicitant sur un exemple ou deux le calcul effectué.

$i \backslash p$	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1									
2									
3									
4									

2. Quelle est la valeur optimale du sac ? Dans quelle case du tableau se lit-elle ?

Exercice 7 *Retrouver les objets emportés* Le tableau T de l'exercice précédent donne la valeur optimale ; mais comment savoir *quels* objets on a effectivement emportés ? Il suffit de remonter le tableau « à l'envers », en partant de la case T[n][W] et en se demandant, à chaque étape, si l'objet a été pris ou non.

On part de la case T[n][W] avec $n = 4$ et W $= 8$, et on applique la règle suivante, pour i de n à 1 :

— si T[i][p] = T[$i-1$][p], c'est que l'objet i n'a pas été pris : on passe à $(i-1, p)$;

— sinon, l'objet i a été pris : on l'ajoute à la solution et on passe à $(i-1, p-p_i)$.

1. Appliquer cet algorithme sur le tableau précédent. Détailler chaque étape.
2. Vérifier que la combinaison trouvée a bien un poids inférieur ou égal à 8 et une valeur totale de 21.

④ Ouverture : la distance d'édition

Le problème. Quand on tape un mot avec une faute, comme `CHATT` au lieu de `CHAT`, le correcteur orthographique cherche le mot du dictionnaire qui ressemble *le plus* à ce qu'on a tapé. Pour mesurer cette ressemblance, on utilise une notion appelée **distance d'édition** (ou *distance de Levenshtein*, du nom de son inventeur en 1965).

La distance d'édition entre deux chaînes A et B est le *nombre minimal* d'opérations élémentaires nécessaires pour transformer A en B. On autorise trois types d'opérations, chacune coûtant 1 :

— **insertion** d'un caractère à une position quelconque ;

— **suppression** d'un caractère existant ;

— **substitution** d'un caractère par un autre.

Quelques exemples pour s'approprier la notion.

— Transformer `CHAT` en `CHATS` : il suffit d'ajouter `S` à la fin. *Distance* = 1 (une seule insertion).

— Transformer `CHAT` en `CHATEAU` : il faut ajouter `E`, `A`, `U` à la fin. *Distance* = 3 (trois insertions).

— Transformer `CHAT` en `CHIEN` : on peut substituer `A`→`I`, substituer `T`→`E`, puis insérer `N` à la fin. *Distance* = 3.

Cette notion est utilisée en bio-informatique (pour comparer des séquences ADN), dans les correcteurs orthographiques (pour suggérer la correction la plus proche du mot tapé), et dans les systèmes de détection de plagiat.

Exercice 8 *Échauffement : trouver la distance à la main*

1. Donner une suite d'opérations qui transforme `NSI` en `NUIT`. Compter le nombre d'opérations utilisées.
2. Peut-on transformer `NSI` en `NUIT` en une seule opération ? Si oui, laquelle ? Sinon, justifier que c'est impossible. En déduire la valeur de la distance d'édition entre `NSI` et `NUIT`.

Exercice 9 *Construire la formule de récurrence* Pour calculer la distance d'édition par programmation dynamique, on va introduire une *famille* de sous-problèmes. Comme dans le sac à dos, c'est en résolvant tous les petits problèmes qu'on résout le grand.

Notation. On fixe deux chaînes A et B. Pour deux entiers i et j, on note :

$d(i, j)$ = distance d'édition entre les i premiers caractères de A et les j premiers caractères de B.

Pourquoi i pour A et j pour B ? On va ranger les valeurs $d(i, j)$ dans un *tableau* (cf. exercice suivant). Or, par convention en mathématiques, i désigne le numéro de *ligne* et j le numéro de *colonne*. On place donc les préfixes de A en lignes (associés à i) et ceux de B en colonnes (associés à j) : ainsi, A « arrive avant B » dans l'alphabet *et* i arrive avant j, ce qui évite d'inverser les conventions en cours de route.

Cas faciles.

— Si $i = 0$ (préfixe vide de A) et j quelconque : pour transformer la chaîne vide en les j premiers caractères de B, il faut j insertions, donc $d(0, j) = j$.

— Symétriquement, $d(i, 0) = i$ (on supprime les i caractères de A pour obtenir la chaîne vide).

Cas général. Supposons $i \geqslant 1$ et $j \geqslant 1$. Pour transformer le préfixe de A (de longueur i) en celui de B (de longueur j), regardons le *dernier caractère* de chaque préfixe. Trois cas, et trois cas seulement, peuvent se produire :

1. **Le dernier caractère de** A **est supprimé** (coût 1 pour cette suppression). Il reste à transformer le préfixe de A de longueur $i - 1$ en celui de B de longueur j : par définition, cela coûte $d(i - 1, j)$.

   Coût total : $d(i - 1, j) + 1$.

   Exemple. Pour passer de A = `CHATS` ($i = 5$) à B = `CHAT` ($j = 4$), on supprime le `S` final de A (coût 1), et il reste le sous-problème `CHAT` → `CHAT` (coût 0). Total : 0 + 1 = 1.

2. **Un nouveau caractère est inséré à la fin de** A**, à savoir le dernier caractère de** B (coût 1 pour cette insertion). Il reste à transformer les i caractères de A en le préfixe de B de longueur $j - 1$: cela coûte $d(i, j - 1)$.

   Coût total : $d(i, j - 1) + 1$.

   Exemple. Pour passer de A = `CHAT` ($i = 4$) à B = `CHATS` ($j = 5$), on insère le `S` final de B à la fin de A (coût 1), et il reste le sous-problème `CHAT` → `CHAT` (coût 0). Total : 0 + 1 = 1.

3. **Le dernier caractère de** A **est mis en correspondance avec le dernier caractère de** B : gratuitement s'ils sont déjà identiques (coût $c = 0$), par substitution sinon (coût $c = 1$). Il reste à transformer le préfixe de A de longueur $i - 1$ en celui de B de longueur $j - 1$: cela coûte $d(i - 1, j - 1)$.

   Coût total : $d(i - 1, j - 1) + c$.

   Exemple 1 (*identité*, $c = 0$). Pour passer de A = `CHAT` à B = `ROT`, les derniers caractères coïncident (`T` et `T`) : rien à faire sur ce caractère, il reste le sous-problème `CHA` → `RO`.

   Exemple 2 (*substitution*, $c = 1$). Pour passer de A = `CHAT` à B = `CHIEN`, les derniers caractères diffèrent (`T` vs `N`) : on substitue `T`→`N` (coût 1), il reste le sous-problème `CHA` → `CHIE`.

Remarque sur la symétrie. Les trois cas correspondent aux trois manières de « réduire » le couple (i, j) : dans le premier, on diminue i (un caractère de A part) ; dans le second, on diminue j (un caractère de B est traité) ; dans le troisième, on diminue les deux à la fois.

On prend la *plus petite* des trois possibilités, d'où la formule :

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

Question : Vérifier que cette formule donne bien $d(0, 0) = 0$, $d(1, 1) = 0$ pour deux chaînes commençant par la même lettre, et $d(1, 1) = 1$ pour deux chaînes commençant par des lettres différentes.

Exercice 10 ***Distance d'édition par tableau*** On veut maintenant appliquer la formule récursive ci-dessus pour calculer la distance d'édition entre A = `CHAT` et B = `CHATS`.

On range les valeurs $d(i, j)$ dans un tableau : l'indice i (lignes) parcourt les préfixes de A, et j (colonnes) ceux de B. La case (i, j) contient $d(i, j)$. Le tableau ci-dessous a sa première ligne et sa première colonne déjà remplies grâce aux *cas faciles* (cf. exercice précédent).

1. Recopier le tableau et le compléter ligne par ligne en utilisant la formule de récurrence. On pourra détailler en marge le calcul d'une ou deux cases.

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

2. Dans quelle case se lit la distance d'édition entre A et B ? Quelle est sa valeur ? Cela correspond-il à ce que vous attendiez ?
3. **Lecture du chemin.** Sur le tableau rempli, partir de la case $(4, 5)$ (en bas à droite) et remonter vers $(0, 0)$ (en haut à gauche), en se déplaçant à chaque étape :

   — d'une case vers le haut (suppression d'un caractère de A) ;

   — d'une case vers la gauche (insertion d'un caractère de B) ;

   — ou d'une case en diagonale haut-gauche (substitution ou identité).

   On choisit le déplacement qui correspond à la valeur la plus petite parmi les trois prédécesseurs. Tracer un tel chemin sur le tableau et lire la suite d'opérations qu'il décrit.

Remarque 3 (Bilan des activités). Nous avons rencontré trois problèmes (Fibonacci, sac à dos, distance d'édition) qui paraissent sans rapport, mais qui partagent tous la même structure :

1. un problème global se décompose en *sous-problèmes* plus petits ;
2. les sous-problèmes se *chevauchent* (on retombe plusieurs fois sur les mêmes) ;
3. il existe une *relation de récurrence* entre la solution du problème et celle des sous-problèmes.

Dès lors, deux stratégies fonctionnent :

— *top-down* avec mémoïsation : on garde l'écriture récursive naturelle, enrichie d'un cache ;
— *bottom-up* avec un tableau : on résout d'abord tous les plus petits sous-problèmes et on remonte.

Dans les deux cas, le nombre d'opérations devient polynomial (au lieu d'exponentiel). C'est ce que l'on appellera la **programmation dynamique**. Le notebook Capytale associé vous permettra d'en explorer les implémentations Python.