Programmation dynamique

1. Définition et idée générale

La programmation dynamique consiste à résoudre un problème en mémorisant les résultats des sous-problèmes déjà calculés afin d'éviter les recalculs inutiles. C'est ce qu'on appelle la **mémoïsation**.

> ✅ **Quand l'employer ?**
>
> La programmation dynamique est surtout utile quand les *mêmes sous-problèmes* reviennent *plusieurs fois*.

> 💭 **Remarque : mémoïsation *vs* mémorisation** ▼
>
> **Mémorisation** est un terme général : c'est le fait de stocker quelque chose en mémoire.
> **Mémoïsation** est un terme technique précis en informatique : c'est une stratégie d'optimisation qui consiste à mettre en cache le résultat d'un appel de fonction selon ses arguments, pour éviter de le recalculer si la fonction est rappelée avec les mêmes arguments. C'est donc une forme particulière de mémorisation.
>
> **En résumé :** toute mémoïsation *est* une mémorisation, mais toute mémorisation **n'**est **pas** une mémoïsation.

2. Applications

La programmation dynamique se distingue de « diviser pour régner » (DPR) :

- en programmation dynamique, les sous-problèmes sont « plus petits », mais ils se *chevauchent* (ou se répètent, se recoupent…) ;
- dans DPR, les sous-problèmes sont « plus petits » **mais** *différents* (…*a priori* : il se *pourrait* qu'ils se recoupent, mais ce serait un *hasard*).

C'est particulièrement visible avec le calcul récursif « naïf » des termes de la suite de Fibonacci (voir ci-dessous).

2.1. Exemple simple : la suite de Fibonacci

La suite de Fibonacci est définie par :

- $F_0 = 0$ et $F_1 = 1$;
- $F_n = F_{n-1} + F_{n-2}$.

Cette définition est *récursive* : on calcule un nouveau terme de la suite en additionnant les deux termes précédents.

> **Version récursive naïve**

```python
def fibo_rec(n):
    print(f"Calcul de F_{n}")
    if n <= 1:
        return n
    else:
        return fibo_rec(n-1) + fibo_rec(n-2)
```

Cette définition récursive est simple à écrire, mais recalcule sans cesse les mêmes valeurs.

✏️ **Exemple : en pratique** ▼

L'instruction `fibo_rec(5)` provoque l'affichage suivant :

```
Calcul de F_5
Calcul de F_4
Calcul de F_3
Calcul de F_2
Calcul de F_1
Calcul de F_0
Calcul de F_1
Calcul de F_2
Calcul de F_1
Calcul de F_0
Calcul de F_3
Calcul de F_2
Calcul de F_1
Calcul de F_0
Calcul de F_1
5
```

Version « programmation dynamique »

```python
def fibo_dyn(n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    print(f"Calcul de F_{n}")
    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fibo_dyn(n - 1, memo) + fibo_dyn(n - 2, memo)
    return memo[n]
```

Cette version utilise une mémoïsation : si `fibo_dyn(n)` a déjà été calculé, on peut réutiliser le résultat sans le recalculer.

✏️ **Exemple : en pratique** ▼

L'instruction `fibo_dyn(5)` provoque l'affichage suivant :

```
Calcul de F_5
Calcul de F_4
Calcul de F_3
Calcul de F_2
Calcul de F_1
Calcul de F_0
5
```

💭 **Remarque : pour aller plus loin (considérations avancées)** ▼

⚠️ **Attention aux arguments par défaut *mutables* !**

En Python, les valeurs par défaut des arguments sont *évaluées* ***une seule fois***, au moment de la définition de la fonction — **pas** à chaque appel. Si cette valeur est un objet *mutable* (liste, dictionnaire…), il est **partagé entre tous les appels** qui n'en fournissent pas explicitement un.

✏️ **Exemple : dans le cas qui nous occupe** ▼

C'est pour cela que l'on trouve `memo=None` dans la définition de `fibo_dyn` : l'initialisation explicite dans le corps est *sûre*.

```python
def fibo_dyn(n, memo=None):
    if memo is None:
        memo = {}   # nouveau dictionnaire à chaque appel "racine"
```

Si on avait écrit :

```python
def fibo_dyn(n, memo={}):  # ⚠️ PIÈGE !
```

…alors *le* ***même*** *dictionnaire serait* ***réutilisé*** *d'un appel à l'autre* .

Cela peut sembler pratique, mais peut provoquer des effets de bord difficiles à déboguer. Ainsi, les résultats d'un premier appel `fibo_dyn(5)` seraient encore dans `memo` lors d'un appel ultérieur `fibo_dyn(10)`.

Dans ce cas précis, ce serait même *bénéfique* — mais c'est un **hasard**, et en général le résultat sera **faux !**

Règle à retenir : ne jamais mettre une liste ou un dictionnaire comme valeur par défaut d'un argument. Utiliser None et initialiser dans le corps de la fonction.

✏️ **Exemple : une situation *vraiment* moisie** ▼

Le code suivant montre une fonction créant *récursivement* la liste des entiers entre `n` et `1` :

```python
def liste_des_entiers(n, resultats=[]):
    if n == 0:
        return resultats
    resultats.append(n)
    return liste_des_entiers(n - 1, resultats)

print(liste_des_entiers(3))  # → [3, 2, 1]  ✅ (premier appel)
print(liste_des_entiers(3))  # → [3, 2, 1, 3, 2, 1]  ⚠️ la liste s'allonge ☠️ !
```

Au deuxième appel, la liste par défaut contient *déjà* `[3, 2, 1]` du premier appel — les nouveaux éléments s'y ajoutent, au lieu de repartir d'une liste vide.

Pour corriger ce code, il suffit de faire :

```
def liste_des_entiers(n, resultats=None):
    if resultats is None:
        resultats = []
    if n == 0:
        return resultats
    resultats.append(n)
    return liste_des_entiers(n - 1, resultats)
```

2.2. Exemple plus avancé

Le problème du sac à dos en version 0/1 se prête très bien à la programmation dynamique (dans cette version « binaire » du problème du sac à dos, « on prend un objet *en totalité*, ou pas du tout » — on *ne* peut *pas* prendre une *fraction* des objets disponibles). En effet, on compare plusieurs choix possibles pour une même capacité restante.

3. Points d'attention complémentaires

- La programmation dynamique évite les recalculs.
- Elle repose sur la mémoïsation (mémorisation des résultats déjà établis).
- Elle sert souvent à obtenir une solution optimale.

> ⚠️ **Différence avec un algorithme glouton**
>
> La programmation dynamique n'est pas la même chose que le glouton : elle ne se contente pas d'un choix *localement optimal* immédiat. Elle explore toutes les combinaisons *possibles* et garantit une solution optimale, là où le glouton peut échouer.

4. Liens avec d'autres notions

- Fibonacci est l'exemple classique à connaître.
- La **récursivité** s'invite volontiers dans le paysage.
- Un algorithme *glouton* peut échouer **là où la programmation dynamique réussit !**
- Le **problème du sac à dos** illustre bien l'intérêt de la méthode (voir la fiche associée).
- La **complexité** est souvent fortement améliorée par rapport à une version naïve.