1 Problématique

Dans un problème récréatif posé en 1202 par Leonardo Fibonacci, on décrit la croissance d'une population de lapins.
« Quelqu'un a déposé un couple de lapins dans un certain lieu, clos de toutes parts, pour savoir combien de couples seraient issus de cette paire en une année, car il est dans leur nature de générer un autre couple en un seul mois, et qu'ils enfantent dans le second mois après leur naissance. [1] »

Mathématiquement cette relation récursive s'écrit :

$$F_n = \begin{cases} F_0 = 0 & \text{si } n = 0 \\ F_1 = 1 & \text{si } n = 1 \\ F_n = F_{n-1} + F_{n-2} & \text{si } n > 1 \end{cases}$$

Comment obtenir un calcul efficace des termes de la suite ?

2 Mise en évidence du problème

L'implémentation immédiate de la relation donne la fonction *fibo* :

```
1  def fibo(n: int)->int:
2      """
3      calcule le terme de rang n
4      de la suite de Fibonacci
5      """
6      if n == 0:
7          return 0
8      elif n == 1:
9          return 1
10     else:
11         return fibo(n-1) + fibo(n-2)
```

L'arbre des appels pour $n = 8$ (figure 1) montre la redondance des calculs.

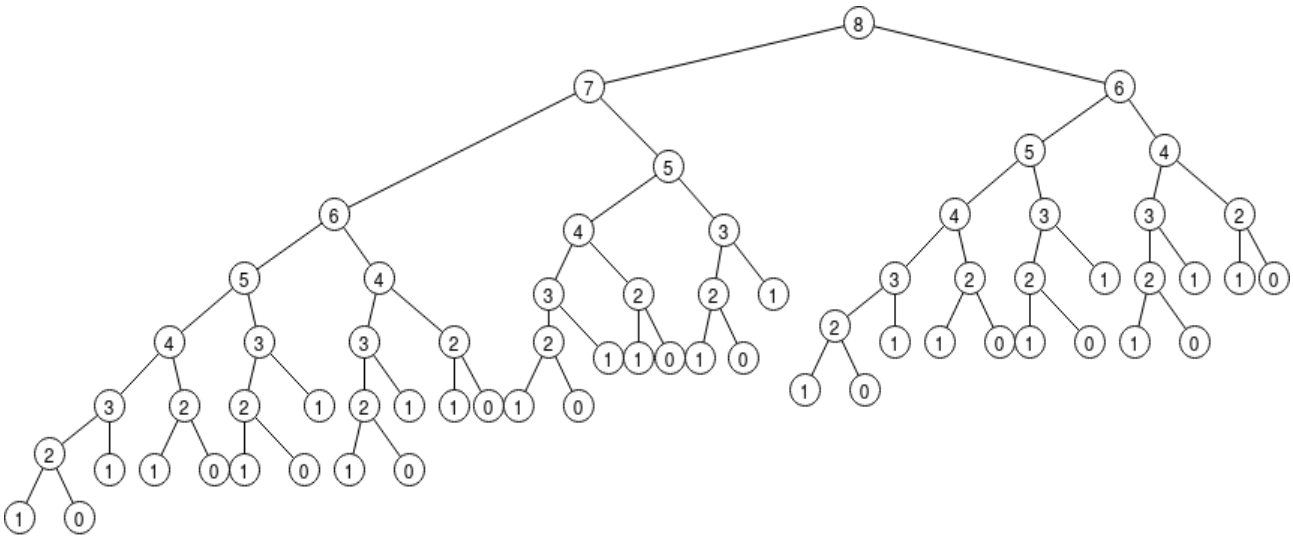

FIGURE 1 – Appels récursifs pour $n = 8$

1. Davantage d'informations sur https://tinyurl.com/wikifibo

Christophe Viroulaud

Activité 1 :

1. Tester la fonction *fibo* pour $n = 8$, $n = 10$.
2. À l'aide d'une variable globale (c'est mal) *compteur*, observer le nombre d'appels réalisés en fonction de n.

3 Programmation dynamique

Le principe de la programmation dynamique peut être considéré comme une amélioration de l'approche *diviser pour régner*. Il consiste à réutiliser des résultats déjà calculés lors du découpage en sous-problèmes.

3.1 Top-down

Cette approche part du problème principal (*Top* → *Haut*) qui est découpé en petits problèmes (*Down* → *Bas*).

```
1  def fibo_top_down(n: int, track: list)->int:
2      """
3      calcule le terme de rang n
4      de la suite de Fibonacci
5      """
6      if track[n] > 0:
7          return track[n]
8      if n == 0:
9          track[0] = 0
10         return track[0]
11     elif n == 1:
12         track[1] = 1
13         return track[1]
14     else:
15         track[n] = fibo_top_down(n-1, track) + fibo_top_down(n-2, track)
16         return track[n]
```

Code 1 – Approche top-down

Le tableau *track* stocke les valeurs déjà calculées pour éviter d'effectuer les appels redondants.

```
1  n = 20
2  track = [-1 for _ in range(n+1)]
3  fibo_top_down(n, track)
```

Code 2 – Création du tableau de mémoïsation

À retenir

La *mémoïsation* consiste à la *mise en cache* les valeurs déjà calculées pour pouvoir être réutilisées.

Christophe Viroulaud

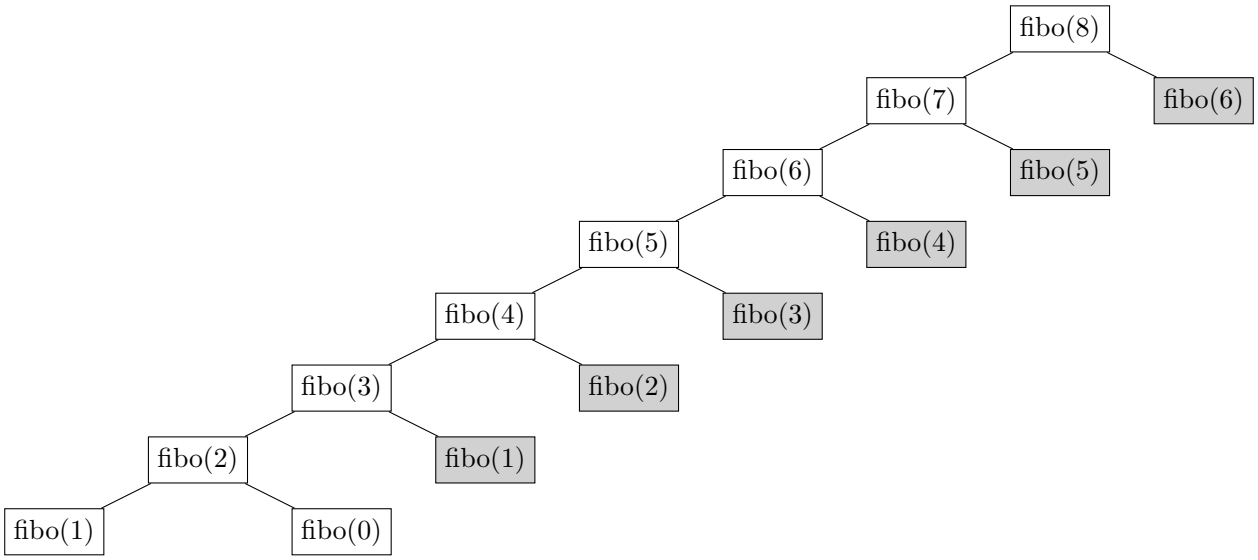

FIGURE 2 – Appels récursifs pour $n = 8$

Activité 2 : Tester la fonction en approche top-down. Compter le nombre d'appels.

3.2 Bottom-up

Cette approche *itérative* résout d'abord les sous-problèmes (*Bottom* → *Bas*) avant de remonter vers le problème principal (*Up -> Haut*).

```
1 def fibo_bottom_up(n: int)->int:
2     track = [0 for _ in range(n+1)]
3     track[1] = 1
4     for i in range(2, n+1):
5         track[i] = track[i-1] + track[i-2]
6     return track[n]
```

Code 3 – Approche bottom-up

On calcule toutes les valeurs en partant de 0.

Activité 3 : Combien d'itérations effectue-t-on ?

top-down ou bottom-up ? complexité en temps souvent identique. Complexité en espace peut varier. S'il est simple de déterminer quels résultats vont être nécessaires, l'approche bottom-up est intéressante → on ne garde que les résultats qui nous intéressent.

Christophe Viroulaud