Programmation dynamique suite de Fibonacci

Christophe Viroulaud

Terminale - NSI

Algo 24

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

Comment obtenir un calcul efficace des termes de la suite ?

Sommaire

Mise en évidence du problème

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

Activité 1 : Écrire la fonction `fibo(n: int)` $\rightarrow$ `int` qui calcule le terme de rang `n` de la suite.

Correction

```
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)
```

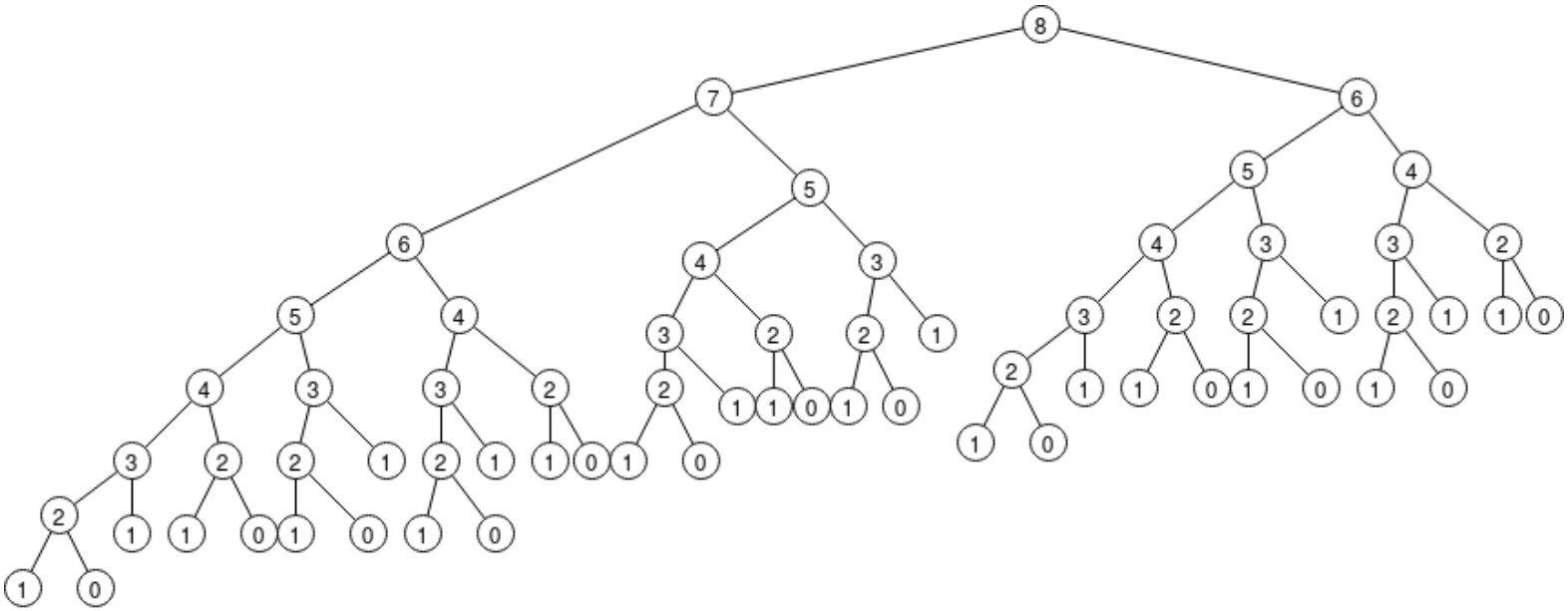

FIGURE 1 – Pour `n = 8`

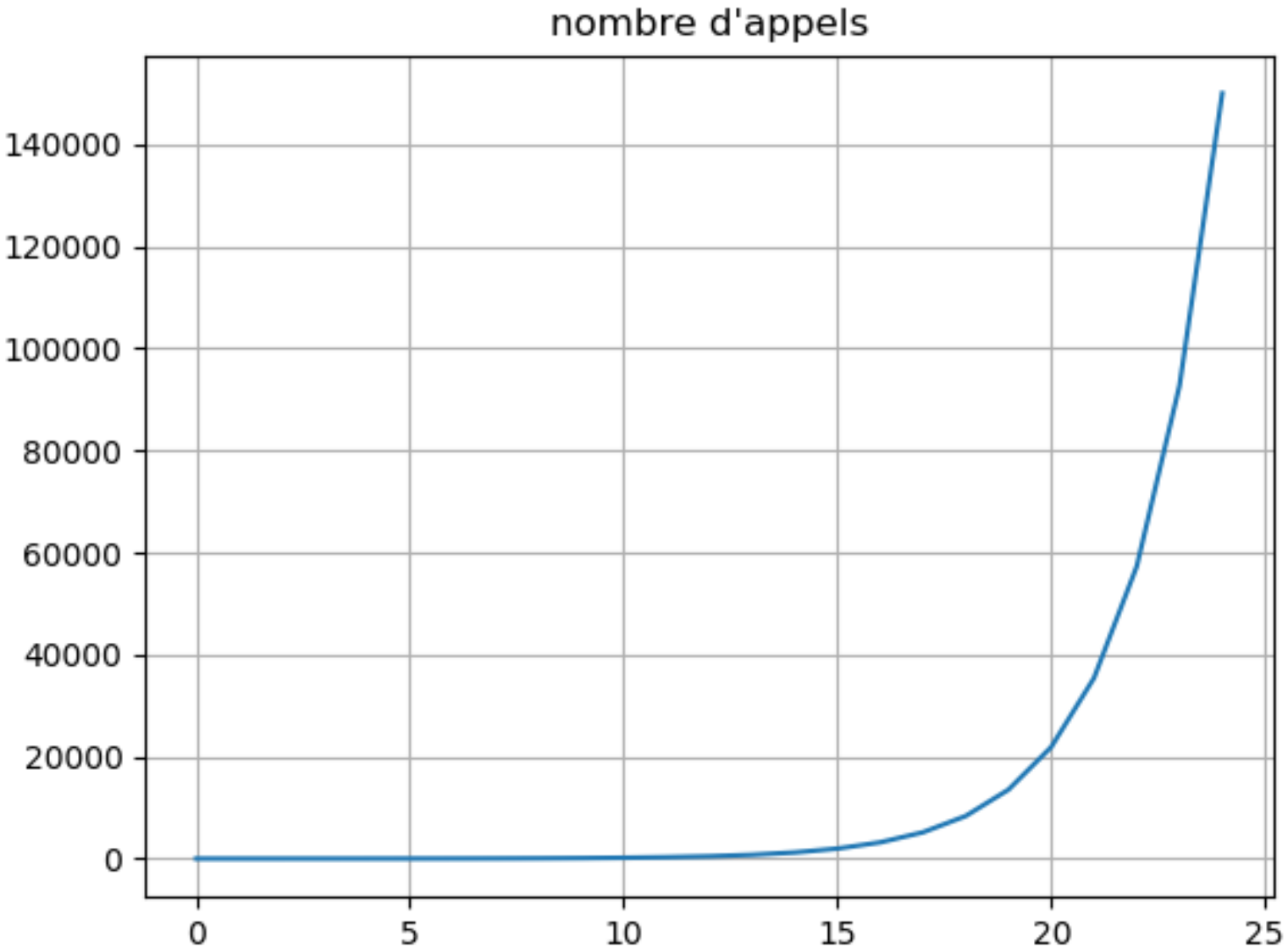

FIGURE 2 – Nombre d'appels en fonction de n

Remarque

La fonction calcule plusieurs fois la même valeur. Le nombre d'appels augmente de manière exponentielle.

Sommaire

Programmation dynamique - Principe

La programmation dynamique :

- s'appuie sur le principe de **diviser pour régner**,
- stocke les résultats intermédiaires pour éviter de les calculer à nouveau.

Remarque

Richard Bellman développe cette stratégie algorithmique dès les années 1950.

Remarque

En toute rigueur, on n'applique pas tout à fait le principe **diviser pour régner** dans la suite de Fibonacci : les problèmes ne sont pas indépendants.

Sommaire

Approche top-down

À retenir

Dans l'approche **top-down (ou descendante)**, on :

- applique une méthode récursive,
- stocke les résultats des appels récursifs (**mémoïsation**).

```
1  def fibo_td(n: int, track: list) -> int:
2      # déjà calculé
3      if track[n] > 0:
4          return track[n]
5      if n == 0:
6          track[0] = 0
7          return track[0]
8      elif n == 1:
9          track[1] = 1
10         return track[1]
11     else:
12         track[n] = fibo_td(n-1, track) + \
13                    fibo_td(n-2, track)
14         return track[n]
```

Code 1 – **track** stocke les résultats déjà calculés.

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

Code 2 – Appel de la fonction

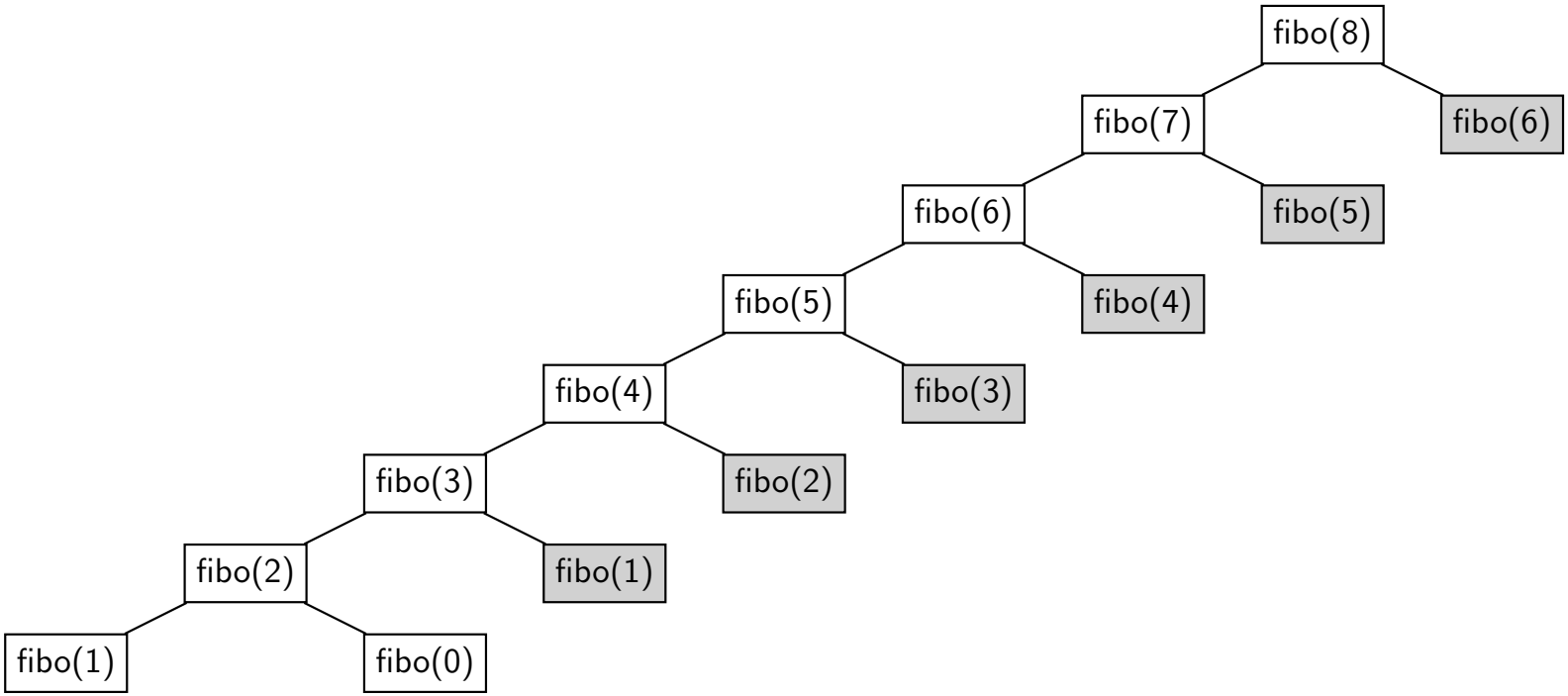

FIGURE 3 – Appels récursifs pour $n = 8$

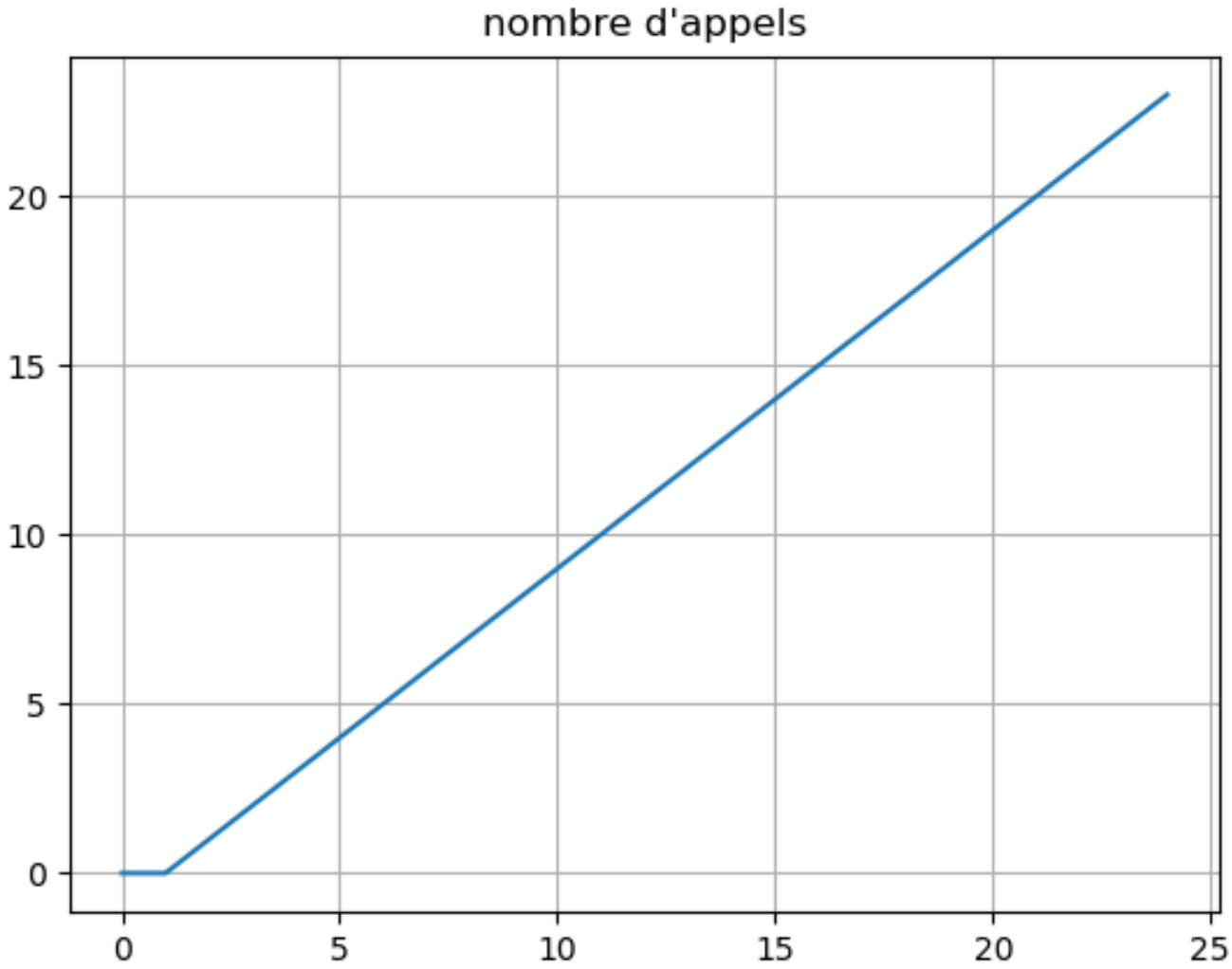

FIGURE 4 – Nombre d'appels en fonction de `n`

Sommaire

Approche bottom-up

À retenir

Dans l'approche **bottom-up (ou ascendante)**, on :

- applique une méthode itérative,
- résout d'abord les petits problèmes.

```python
def fibo_bu(n: int) -> int:
    # résultats déjà calculés
    track = [0 for _ in range(n+1)]
    track[1] = 1

    # calcule de proche en proche
    for i in range(2, n+1):
        track[i] = track[i-1] + track[i-2]

    return track[n]
```

Code 3 – **track** stocke les résultats.

À retenir

La complexité en temps de cette approche est équivalente à celle précédente : on ne calcule chaque sous-problème qu'une seule fois.

Sommaire

Optimisation

Remarque

La complexité en espace dépend de la taille du tableau `track`. Cependant, si seul le résultat final nous intéresse, il est possible d'optimiser l'approche ascendante.

```python
def fibo_bu(n: int) -> int:
    track0 = 0
    track1 = 1

    for i in range(2, n+1):
        # calcule de proche en proche
        track0, track1 = track1, track0 + track1

    return track1
```

Code 4 – On ne garde en mémoire que les deux dernières valeurs.