TP #1

Approche de la programmation dynamique sous Python

La programmation dynamique permet d'optimiser la résolution de problèmes. Elle repose sur le mécanisme de division d'un problème à résoudre en plusieurs sous-problèmes dépendants et à stocker, selon le principe de mémoïsation, les résultats intermédiaires correspondant aux sous-problèmes traités pour ensuite établir la solution du problème initial. L'objet de ce TP consiste alors à appréhender l'approche de la programmation dynamique et sa mise en œuvre à l'aide du langage Python.

Comparaison des approches récursive et dynamique

L'approche de programmation dynamique est souvent confrontée à celle de la programmation récursive. En considérant le cas classique et courant du calcul du $n^{ième}$ terme de la suite de Fibonacci, il est simple de comparer les deux approches et ainsi de montrer l'intérêt principal de la programmation dynamique.

L'implémentation de la fonction `fibo()` en programmation récursive et son appel avec pour argument n=5 sont alors présentés ci-dessous :

```
def fibo(n):
    if n < 0:
        print("Valeur incorrecte !")
    elif n == 0:
        return 0
    elif n == 1 or n == 2:
        return 1
    else:
        return fibo(n - 1) + fibo(n - 2)

print(fibo(5))
```

Cette fonction est tout à fait correcte du point de vue algorithmique et fonctionnel. Elle permet en effet de calculer et d'établir le résultat attendu correspondant au terme passé en argument. Cependant, elle apparaît comme étant peu efficace en considérant que plusieurs valeurs de la suite de Fibonacci sont calculées plusieurs fois pour aboutir au résultat.

La figure 1 montre l'arbre binaire d'évaluation récursive pour n=5. Elle décompose l'enchaînement des traîtements successifs effectués par la fonction `fibo()` et met en évidence le fait que certains termes de la suite de Fibonacci sont effectivement calculés à plusieurs reprises. Par exemple, l'évaluation pour n=2 est exécutée trois fois et celle pour n=3 est effectuée deux fois. Au total, quinze appels à la fonction `fibo()` sont nécessaires au calcul du $5^{ième}$ terme.

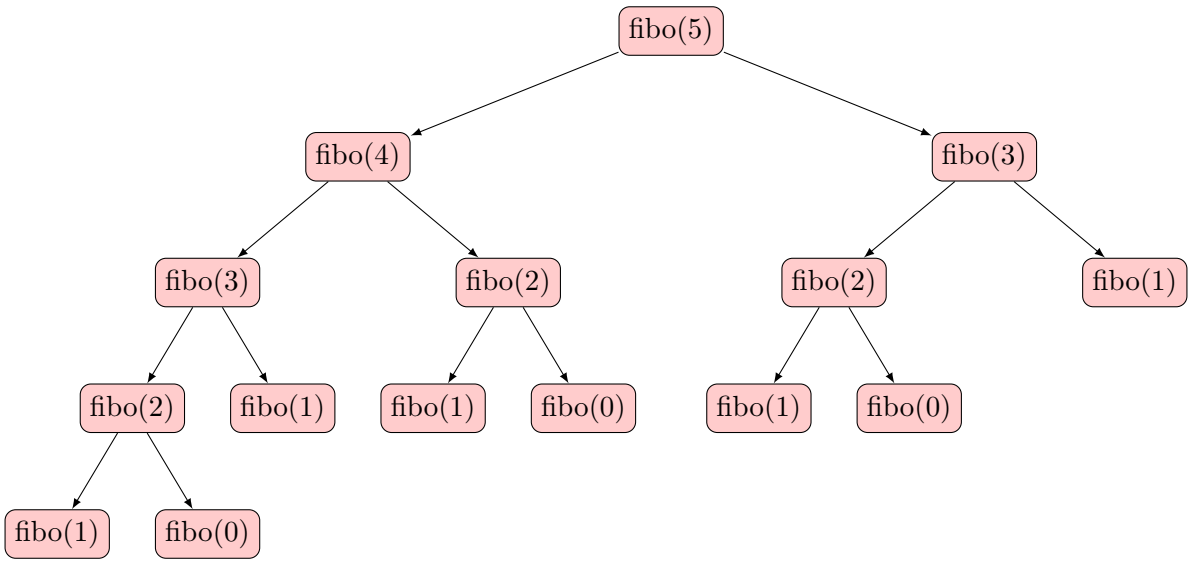

Fig. 1 – *Arbre binaire d'évaluation récursive*

La programmation dynamique constitue alors alternative pertinente et plus performante du calcul des termes de la suite de Fibonacci. Chaque terme de la suite de Fibonacci n'est ainsi calculé qu'à une seule reprise et ensuite mémorisé pour être rappelé en cas de nécessité.

Implémentation selon l'approche de la programmation dynamique

Du point de vue de l'implémentation en Python, le mécanisme de mémoïsation peut aisément reposer sur l'utilisation d'une structure de données. Le stockage des différents termes calculés, et donc des solutions intermédiaires, peut ainsi être assuré au moyen d'une liste.

En cas de besoin, la fonction de calcul de la suite de Fibonacci doit alors vérifier l'existence préalable dans la liste du terme concerné afin de ne pas réitérer inutilement son calcul.

Le bout de code ci-après rappelle la façon avec laquelle une liste vide peut être déclarée et la manière d'y ajouter des éléments :

```
memo = []

for i in range(20):
    memo.append(1)
```

Une autre façon plus synthétique de procéder consiste est présentée ci-dessous :

```
memo = [1] * 20
```

Dans les deux exemples ci-dessus, une liste nommée `memo` est déclarée et vingt éléments de type entier lui sont ajoutés. Chaque élément est ensuite initialisé avec la valeur *1*.

Travail à réaliser

1. Concevoir et écrire un programme Python permettant de calculer le $n^{ième}$ terme de la suite de Fibonacci en programmation dynamique. Une liste nommée `memo` doit être utilisée pour répondre au mécanisme de mémoïsation appliqué pour le stockage de la valeur des différents termes.
2. Instrumenter et exécuter le code de la fonction implémentée afin de permettre l'évaluation du temps nécessaire au calcul. L'utilisation du module Datetime met à disposition les éléments de programmation nécessaires à l'évaluation du temps comme le montre le bout de code suivant :

```
import time

start = time.process_time()
...
stop = time.process_time()
print('Délai : ' + str((stop - start) * 1000000) + ' µs')
```

3. Instrumenter et exécuter le code fourni concernant la programmation itérative afin de permettre l'évaluation du temps nécessaire au calcul.
4. Mettre graphiquement en forme les résultats d'évaluation temporelle à l'aide de la bibliothèque Matplotlib [1] et en déduire la complexité algorithmique temporelle pour chacune des deux approches de programmation. La figure 2 montre un exemple de report des résultats obtenus dans le cas de la programmation récursive et en considérant une configuration logarithmique de l'axe du temps.

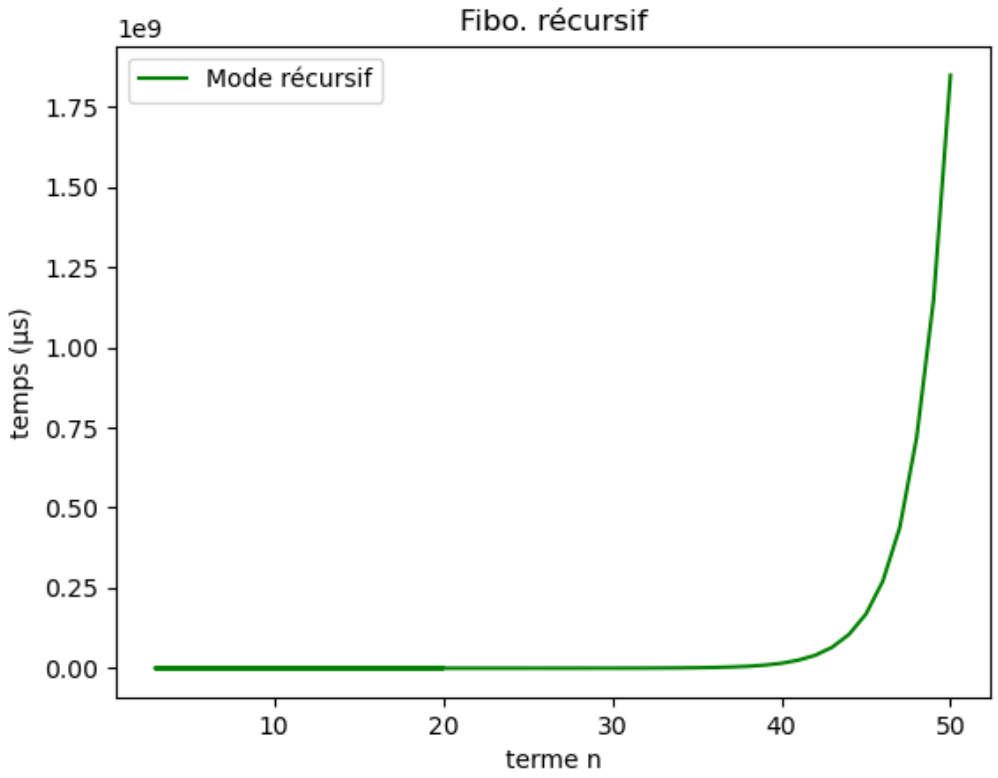

Fig. 2 – *Graphique de report des temps d'exécution*

5. Comparer les résultats d'évaluation temporelle obtenus par les deux fonctions instrumentées pour le calcul de quelques dizaines de termes de la suite de Fibonacci et conclure sur le gain de performance obtenu par l'implémentation en programmation dynamique.

1. https ://matplotlib.org