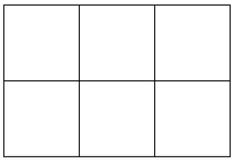
Chapitre 17 - Programmation dynamique

Objectifs :

- ▷ Utiliser la programmation dynamique pour écrire un algorithme.

1 Introduction

On dispose de la grille 2×3 ci-dessous.



Question : Combien de chemins mènent du coin supérieur gauche au coin inférieur droit, en se déplaçant uniquement le long des traits horizontaux vers la droite et le long des traits verticaux vers le bas ?

Et pour une grille 10×10 ?

2 Principe de la programmation dynamique

A retenir !

La programmation dynamique est une technique dûe à *Richard Bellman* dans les années 1950. À l'origine, cette méthode algorithmique était utilisée pour résoudre des problèmes d'**optimisation**.

L'idée générale est de déterminer un résultat sur la base de calculs précédents.

Plus précisément, la programmation dynamique consiste à résoudre un problème :

— en le **décomposant en sous-problèmes**,

— puis à **résoudre les sous-problèmes** des plus petits au plus grands

— **en stockant les résultats intermédiaires**.

Le terme programmation désigne la **planification**, et n'a pas de rapport avec les langages de programmation.

3 La suite de Fibonacci

3.1 Rappels

On a déjà abordé cette suite lorsque nous avons parlé de la programmation récursive (Chapitre 1).

La **suite de Fibonnacci** est une suite de nombres dont chacun est la somme des deux précédents. Le premier et le second nombres sont égaux à 0 et 1 respectivement.

On obtient la suite de nombres : 0 - 1 - 1 - 2 - 3 - 5 - 8 - 13 - 21 - ..., *etc.*

Mathématiquement, cette suite notée F_n est définie par :

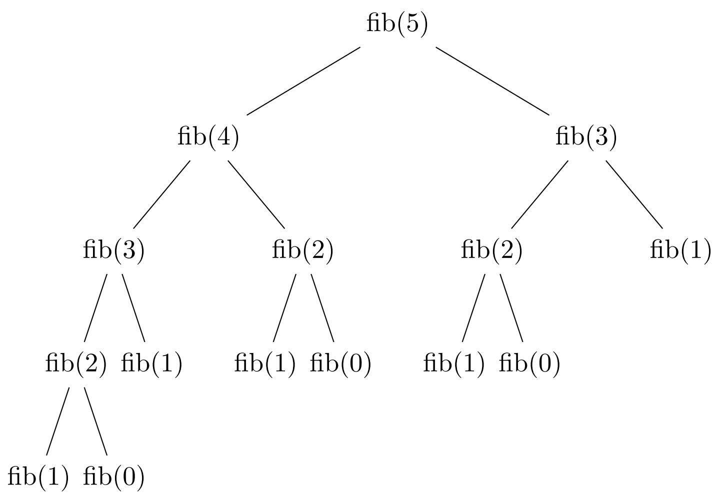
$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ pour tout entier } \geq 2 \end{cases}$$

3.2 Version récursive naïve (et inefficace)

Nous avons déjà programmé une version récursive qui renvoie le terme de rang n de cette suite.

```python
def fib0(n):
    """Version recursive naive"""
    if n <= 1:
        return n
    else:
    return fib0(n-1) + fib0(n-2)
```

Par exemple, voici l'arbre des appels récursifs si on lance `fibo(5)`.



On se rend compte qu'il y a beaucoup d'appels redondants :

- `fib(1)` a été lancé 5 fois,
- `fib(2)` a été lancé 3 fois,
- *etc.*

Ces redondances entraînent un nombre d'appels récursifs qui explose rapidement dès que n est élevé. Par conséquent, les temps de calcul deviennent vite très élevés. Pire, dès que n est trop grand, l'algorithme ne donnera jamais la réponse.

Par exemple, en utilisant un programme principal qui calcule et affiche le temps d'exécution :

```python
start_time = time.time()                # Debut du chronometre
fibo(25)                                # Code a mesurer
end_time = time.time()                  # Fin du chronometre

execution_time = end_time - start_time       # Calcul du temps ecoule
print("Temps d'execution :", execution_time, "secondes")
```

On obtient :

- si on exécute `fibo(25)`, temps d'exécution : `30 ms`
- si on exécute `fibo(35)`, temps d'exécution : `3.62 s`
- si on exécute `fibo(40)`, temps d'exécution : `36.77 s`

A retenir !

Il est possible de faire mieux, en évitant de refaire les calculs déjà effectués.

Pour cela, il faut **stocker les résultats intermédiaires** !

4 Version récursive avec mémoïsation

Une première approche est d'adapter l'algorithme récursif **en stockant les résultats calculés dans un tableau ou un dictionnaire**.

Lors d'un appel, on commence par vérifier si on ne connaît pas déjà la réponse, auquel cas on la renvoie directement, ce qui évite d'effectuer des calculs redondants.

Cela donne la fonction `fibo_memo` suivante qui prend en paramètres un entier `n` et un dictionnaire `memo` que l'on met à jour en stockant les résultats intermédiaires au fur et à mesure.

```python
1 def fibo_memo(n,memo):
2     if n in memo:                # si calcul deja effectue
3         return memo[n]           # on renvoie directement sa valeur
4     elif n <= 1:
5         memo[n] = n
6         return memo[n]
7     else:
8         memo[n] = fibo_memo(n-1,memo) + fibo_memo(n-2,memo)
9         return memo[n]
```

Explications :

- **Lignes 2 et 3 :** si la valeur n est déjà dans le dictionnaire, c'est qu'on a déjà calculé F_n et il suffit alors de renvoyer sa valeur `mem[n]` (la valeur associée à `n`).
- **Lignes 4 à 9 :** quasiment identiques à la version récursive naïve à ceci près que l'on mémorise la valeur dans le dictionnaire `memo` avant de la renvoyer.
- De cette façon, dès qu'une valeur F_n a été calculée, elle est ajoutée dans le dictionnaire comme la valeur associée à `n`, ce qui permet de la réutiliser directement dès qu'on en a besoin.

Il n'y a plus qu'à lancer le premier appel avec un dictionnaire vide, c'est ce que fait la fonction fibo suivante.

```python
def fibo(n):
    """Version recursive avec memoisation"""
    F = {}
    return fibo_memo(n,F)
```

Avec ce procédé de mémoïsation, l'arbre des appels est considérablement réduit puisqu'il n'y a plus aucun appel redondant.

Par exemple, l'arbre des appels récursifs en lançant `fibo(5)` se réduit à :