	**Algorithmique**	**NSI T^{ale}**
	Programmation dynamique	Cours/TD

*

1. Stratégies algorithmiques

Nous avons déjà abordé les **stratégies algorithmiques** à deux reprises :

- **Algorithme glouton** : construit une solution de manière incrémentale, en optimisant un critère de manière locale.

- **Diviser pour régner** : divise un problème en sous-problèmes indépendants, résout chaque sous-problème, et combine les solutions des sous-problèmes pour former une solution du problème initial.

Nous allons donc découvrir une troisième stratégie :

- **Programmation dynamique** : divise un problème en sous-problèmes qui sont non indépendants, et cherche (et stocke) des solutions de sous-problèmes de plus en plus grands

2. Bref historique

Programmation dynamique :
Stratégie développée par **Richard Bellman** en **1953** chez RAND Corporation.

Ici « Programmation » signifie planification :

- **Technique de conception d'algorithme très générale et performante**.
- Permet de résoudre de nombreux **problèmes d'optimisation**.

Richard Bellman

3. Pourquoi « programmation dynamique » ?

« The 1950s were not good years for mathematical research.
We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word 'research'. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term 'research' in his presence. You can imagine how he felt, then, about the term 'mathematical'.
The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation.
What title, what name, could I choose? »

Richard Bellman (1984)

4. Revisitons Fibonacci...

Soit F_n le nombre de lapins au mois n :

$F_1 = 1$

$F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$

Ce sont les **nombres de Fibonacci** :

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Ils croissent très vite (chaque nombre est la somme des deux précédents) :

$F_{30} > 10^6$!!!!!

En fait, $F_n \approx 2^{0.694n}$, ce qui montre une croissance exponentielle.

Leonardo da Pisa, dit Fibonacci

4.1. Algorithme récursif inefficace

D'après la définition, on trouve très rapidement l'algorithme récursif suivant :

```
fonction fib1_rec(n)
    si n = 1 alors retourner 1
    si n = 2 alors retourner 1
    retourner fib1_rec(n-1) + fib1_rec(n-2)
```

→ Implémenter cette fonction en Python et étudier ses performances grâce au module timeit par exemple en utilisant le code suivant :

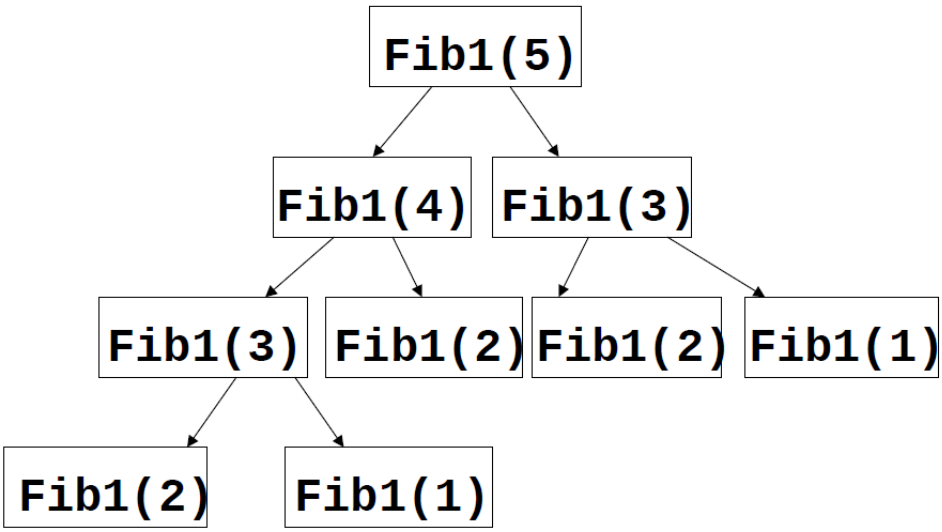
```
from timeit import timeit
import matplotlib.pyplot as plt

abcisse = [5,10,15,20,25,28,30]
ordonnee=[0 for n in range(len(abcisse))]
for i in range(len(abcisse)) :
    ordonnee[i] = timeit("fib1_rec(abcisse[i])", number=10,
                                                 globals=globals())

# Graphique pour le tri sélection en rouge
plt.plot(abcisse, ordonnee, "ro-") # en rouge

plt.show()
plt.close()
```

Pourquoi cette augmentation très rapide du temps d'exécution ?
Si on analyse pour n = 5 on remarque qu'il faut réaliser tous ces appels :



4.2. Algorithme récursif terminal

On peut rendre l'algorithme récursif précédent nettement plus efficace en le rendant terminal :

```
fonction fib1_term(n, terme1=0, terme2=1)
    si n = 0 alors retourner terme1
    sinon retourner fib1_term(n-1, terme2, terme1 + terme2)
```

Cette version correspond à la version itérative optimisée vue plus tard.

4.3. Algorithme récursif avec mémoïsation

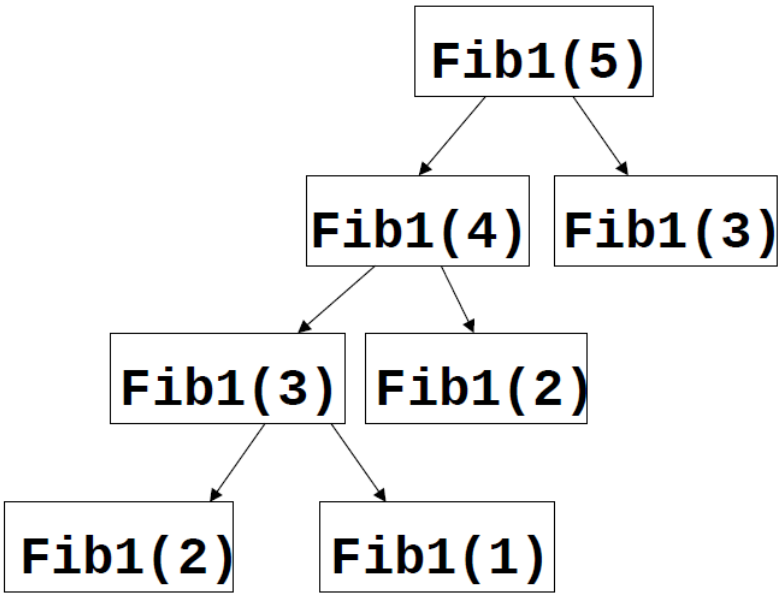
Mémoïser c'est **conserver** à la fin de l'exécution d'une fonction le résultat associé aux arguments d'appels, pour ne pas avoir à recalculer ce résultat lors d'un autre appel récursif.

Cette fois-ci on va sauvegarder à chaque étape le résultat calculé auparavant, ce qui est une bonne idée dans ce cas :

```
fonction fib1_mem(n)
    si mémo[n] est défini alors retourner mémo[n]
    si n <= 2 alors F = 1
    sinon F = fib1_mem(n-1) + fib1_mem(n-2)
    mémo[n] = F
    retourner F
```

→ Implémenter cette fonction en Python et étudier ses performances grâce au module timeit sur les 2000 premières valeurs

Ici le temps de calcul est grandement amélioré car le nombre de calculs est réduit :



4.4. *Analyse de complexité*

Pour la fonction récursive « naïve » :

```
fonction fib1_rec(n)
    si n = 1 alors retourner 1
    si n = 2 alors retourner 1
    retourner fib1_rec(n-1) + fib1_rec(n-2)
```

R(n) le nombre d'appels récursifs pour calculer `fib1_rec(n)` :

$$R(n) = \begin{cases} 1 \text{ si } n = 0,1 \\ R(n-1) + R(n-2) \end{cases}$$

Suite récurrente linéaire d'ordre 2 :

$$R(n) = \frac{1+\sqrt{5}}{2\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

Cette complexité est exponentielle, autrement dit, très mauvaise !

Pour la fonction avec mémoïsation :

```
fonction fib1_mem(n)
    si mémo[n] est défini alors retourner mémo[n]
    si n <= 2 alors F = 1
    sinon F = fib1_mem(n-1) + fib1_mem(n-2)
    mémo[n] = F
    retourner F
```

La fonction `fib1_mem(k)` induit des appels récursifs seulement la première fois qu'elle est appelée.
Nombre d'appels non mémoïsés : n.
Temps d'un appel (sans compter les appels récursifs non mémoïsés) : O(1) du fait de l'addition entière `Fib1mem(k-1) + Fib1mem(k-2)`.
(`mémo[i]` est retourné en Θ(1) si `mémo` est un tableau.)

Complexité temporelle : Θ(n).

4.5. *Plus généralement*

L'idée de la mémoïsation est de réutiliser les solutions de sous-problèmes qui aident à résoudre le problème principal. Elle fait donc partie de la programmation dynamique.

Complexité temporelle : nombre de sous-problèmes x complexité par sous-problème*
(* : on ne compte pas les appels récursifs)

4.6. *Approche « du bas vers le haut »*

Dans cette approche, on remplit un tableau :

```
fonction fib2_bh(n)
    Créer un tableau fib[1..n]
    fib[1] = 1
    fib[2] = 1
    pour i = 3 à n:
        fib[i] = fib[i-1] + fib[i-2]
    retourner fib[n]
```

→ Implémenter cette fonction en Python et étudier ses performances grâce au module timeit sur les 2000 premières valeurs

On effectue les mêmes calculs que dans la version mémoïsée. **Il faut toutefois identifier un ordre** dans lequel résoudre les sous-problèmes.
Complexité temporelle : nombre de cellules du tableau x complexité de calcul d'une cellule.

Cette approche permet souvent de baisser la complexité spatiale.

On peut d'ailleurs l'améliorer encore en ne stockant pas les résultats intermédiaires :

```
fonction fib2_opt(n)
    fib1 = 1
    fib2 = 1
    pour i = 3 à n:
        temp = fib2
        fib2 = fib1 + fib2
        fib1 = temp
    retourner fib2
```

La complexité spatiale de `fib2_bh` : $O(n^2)$ car on a un tableau de n cases avec des entiers codés sur au plus n bits.
La complexité spatiale de `fib2_opt` : O(n) car les variables `fib1` et `fib2` comportent des entiers codés sur au plus n bits.

Ces deux algorithmes sont une approche du **bas vers le haut** et font partie de la **programmation dynamique**.

5. Concevoir une procédure de programmation dynamique

Quatre étapes :

- Définir les **sous-problèmes**.
- Identifier une **relation de récurrence** entre les solutions des sous-problèmes.
- En déduire un **algorithme** récursif avec mémoïsation ou une approche du bas vers le haut
- Résoudre le **problème original** à partir des solutions des sous-problèmes