Programmation dynamique

Introduction :

La programmation dynamique est une approche de résolution de problèmes qui consiste à décomposer un problème complexe en sous-problèmes plus simples et à faire en sorte de ne résoudre qu'une seule fois chaque sous-problème quand celui-ci se répète.

Nous utiliserons tout au long de ce cours la suite de Fibonacci comme fil conducteur pour l'étude de la programmation dynamique. Nous montrerons dans une première partie les limites de la récursivité simple. Dans une deuxième partie nous présenterons les principes généraux et les caractéristiques de la programmation dynamique, que nous mettrons en œuvre dans une troisième partie selon deux approches distinctes, appliquées aux nombres de Fibonacci.

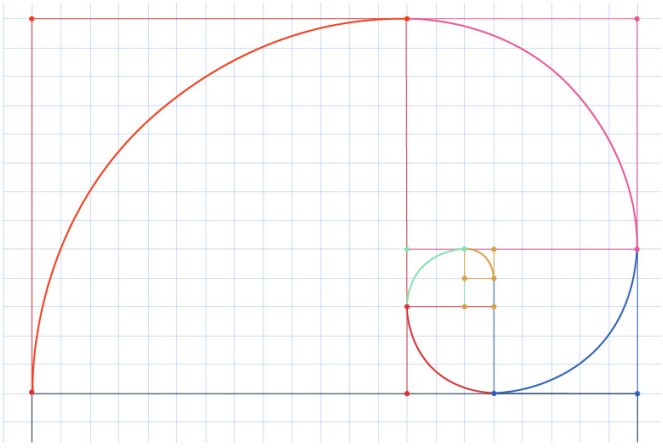
1 Limites de la version récursivité simple

a. Suite de Fibonacci

La suite de Fibonacci est à la fois un exemple classique de programmation récursive et une bonne illustration des limites de la récursivité.

Rappel

La suite de Fibonacci est une suite de nombre entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent, les deux premiers termes étant 0 et 1.



Source : Aloe Polyphylla ©JustChaos

Autrement dit :

- $u(0) = 0$
- $u(1) = 1$
- et $u(n) = u(n-1) + u(n-2)$ pour tout $n > 1$

L'implémentation récursive de la définition mathématique est très naturelle.

b. Implémentation récursive

La définition mathématique de la suite de Fibonacci se transpose très littéralement en algorithme récursif.

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

On obtient facilement la liste des premiers nombres.

```
for i in range(11):
    print(i, fibonacci(i))
```

affiche ceci :

```
0 0
1 1
2 1
3 2
4 3
5 5
6 8
7 13
8 21
9 34
10 55
```

Le calcul et l'affichage des nombres sont quasiment instantanés. Toutefois on constate que même pour des nombres relativement petits, le temps de calcul augmente très rapidement. On peut le constater avec le calcul des nombres de Fibonacci jusqu'à $n = 35$.

```
for i in range(1, 36):
    print(i, fibonacci(i))

# affiche progressivement les termes successifs de la suite jusqu'à 35 inclus.
```

À partir de $n = 30$, le ralentissement de l'affichage devient perceptible.

c. Limites de la solution

S'il ne faut que quelques millisecondes pour calculer les trente premiers nombres de la suite, on observe ensuite un ralentissement de plus en plus marqué de l'affichage des résultats.

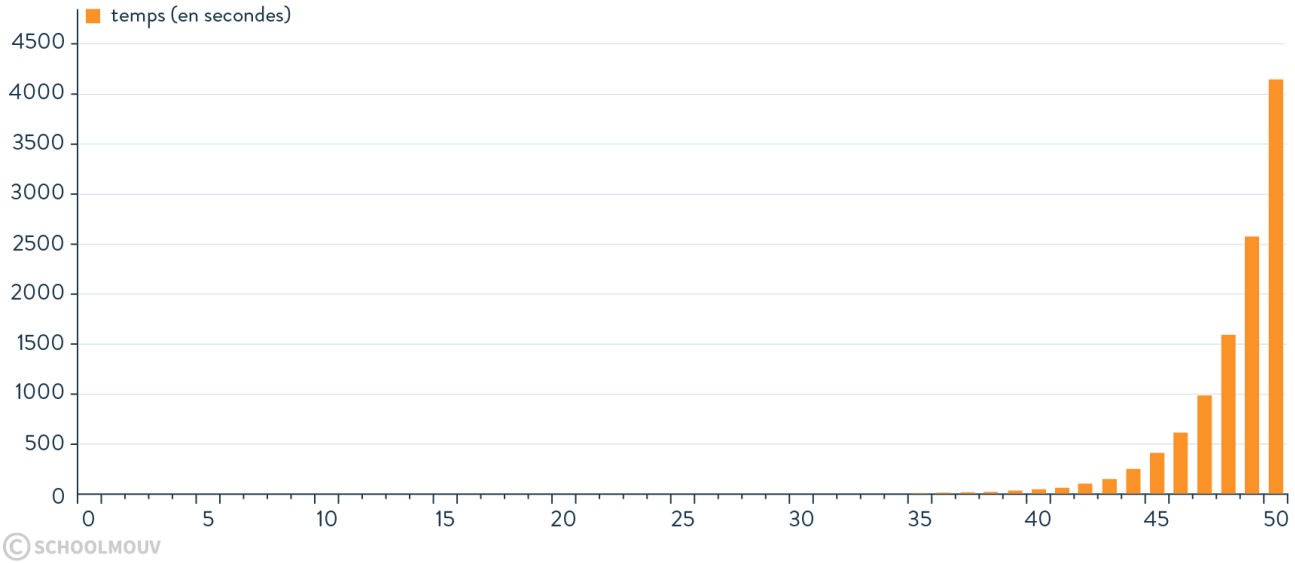
Le calcul des 50 premiers nombres de la suite nous permet de constater à quel rythme les performances de notre algorithme se dégradent.

Il faut environ :

- une seconde pour calculer le 33^{e} terme de la suite ;

- plus de dix secondes pour en calculer le 38^{e} terme ;
- plus de trente secondes pour en calculer le 40^{e} terme ;
- une minute et demie environ pour le 42^{e} terme ;
- dix minutes pour le 46^{e} terme ;
- plus d'une heure pour le 50^{e} terme.

Au total, calculer les 50 premiers nombres de la suite nécessite environ trois heures de calcul avec un ordinateur moderne. Le graphique ci-après illustre le temps de calcul nécessaire.



On constate que les temps de calcul des valeurs de la suite forment une courbe exponentielle.

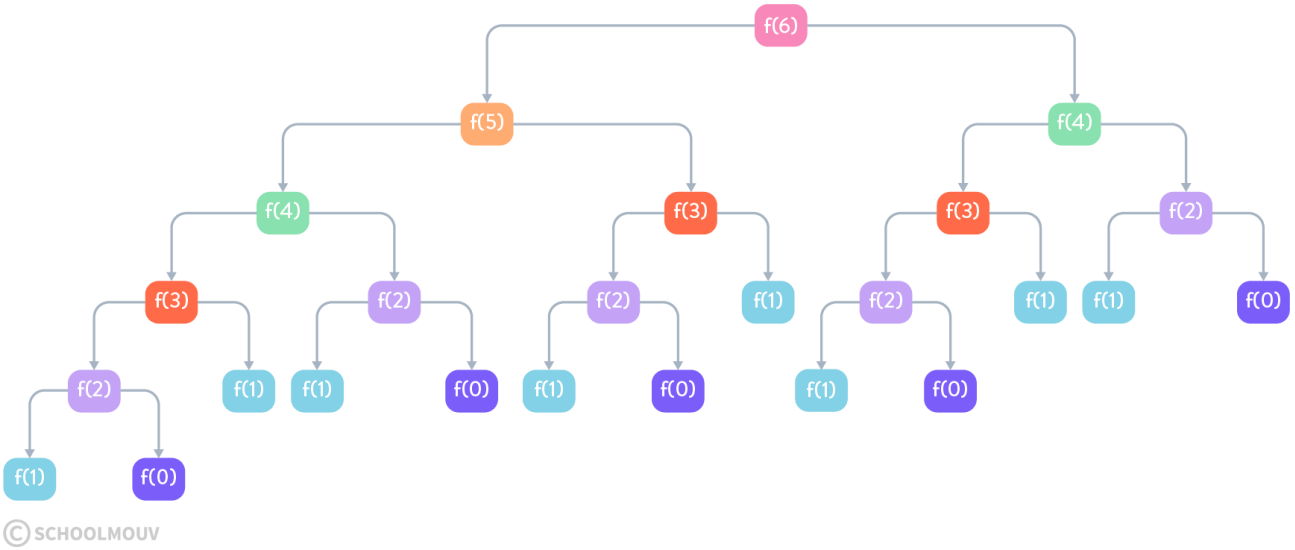
Les nombres eux-mêmes ne sont pourtant pas très grands. Ainsi `fibonacci(50)` vaut seulement $12\ 586\ 269\ 025$, mais il faut plus d'une heure pour le déterminer. Essayons de comprendre ce qui peut ralentir à ce point un ordinateur.

d. Analyse du problème

Intéressons-nous aux appels de fonction récursifs effectués. Un nombre de Fibonacci étant la somme des deux nombres qui le précèdent, son calcul

fait référence à deux nombres qui doivent eux-mêmes être calculés ou retournés.

Prenons le cas du calcul du nombre de Fibonacci pour $n = 6$. Le résultat est 8. Mais combien d'appels de fonctions sont nécessaires pour le déterminer ?
Nous représentons ces appels sous forme d'arbre pour mieux les visualiser et les analyser.



Chaque appel impliquant deux calculs, on observe un doublement du nombre de nœuds à chaque nouveau niveau intermédiaire.

Astuce

> Le nombre d'appels nécessaires pour calculer un nombre de Fibonacci relativement petit augmente très vite : la complexité de cet algorithme étant **exponentielle**, les limites de la machine sont vite atteintes, ainsi qu'on a pu le constater expérimentalement avec des nombres relativement petits.

En analysant cet arbre des appels, on remarque aussi qu'un même calcul est demandé à plusieurs reprises. Par exemple :

- `fibonacci(4)` est appelé deux fois ;
- `fibonacci(3)` est appelé trois fois ;
- `fibonacci(2)` est appelé cinq fois.

Il est raisonnable de penser que les nombreux appels de fonction réitérés pour une même valeur ne constituent sans doute pas une solution optimale.

→ La **programmation dynamique** peut nous permettre de l'optimiser.

2 Principes généraux de la programmation dynamique

a. Origine et applications

La programmation dynamique trouve ses origines dans les années 1950. C'est une méthode d'optimisation mathématique et informatique imaginée par le mathématicien et informaticien américain Richard Ernest Bellman, qui travaillait alors pour RAND Corporation. À cette époque, « programmation » s'entend dans le sens « planification ».
Cette méthode propose une planification dynamique qui permet de résoudre de nombreuses problématiques d'optimisation.

Exemple

La programmation dynamique a été appliquée dans de nombreux contextes autres que l'informatique, comme :

- l'ingénierie, notamment dans le domaine aérospatial ;
- l'économie, notamment pour la gestion des stocks ;
- la bio-informatique, notamment pour le séquençage de l'ADN.

La programmation dynamique repose sur la décomposition d'un problème complexe en sous-problèmes plus simples, avec la particularité que certains de ces sous-problèmes se recoupent.

Définition

Programmation dynamique :

> La programmation dynamique est un paradigme qui consiste à décomposer un problème en sous-problèmes et à conserver les résultats obtenus pour pouvoir les réutiliser.

À retenir

> La programmation dynamique fait en sorte d'éviter d'avoir à résoudre de manière répétitive un même sous-problème.

Ce caractère répétitif des sous-problèmes fait partie des caractéristiques justifiant la programmation dynamique.

b. Principales caractéristiques

La programmation dynamique repose sur deux caractéristiques complémentaires du problème à résoudre :

- la notion de sous-structure optimale ;
- le chevauchement des sous-problèmes.

→ Un problème présente une sous-structure optimale si une solution optimale peut être obtenue à partir des solutions optimales de ses sous-problèmes.

→ Un problème présente des sous-problèmes récurrents s'il peut être divisé en sous-problèmes et que cette décomposition en sous-problèmes entraîne des chevauchements.

Attention

> La décomposition du problème en sous-problèmes est également pratiquée dans le cadre de l'approche « diviser pour régner » qui fait l'objet d'un cours distinct.

La principale différence entre les deux approches réside dans l'indépendance ou non des sous-problèmes :

- dans le paradigme de programmation dynamique, les sous-problèmes ne sont pas indépendants, ils se chevauchent ;

- dans le paradigme « diviser pour régner » les problèmes sont indépendants les uns des autres et ne se répètent pas.

À retenir

> - L'optimisation algorithmique par programmation dynamique n'est possible que si certains sous-problèmes se chevauchent.
> - Si les sous-problèmes ne se répètent pas, il n'y a aucun intérêt à conserver leurs résultats intermédiaires puisqu'ils ne feront l'objet d'aucune réutilisation ultérieure.

c. Mise en œuvre

La mise en œuvre des principes de programmation dynamique peut s'effectuer de deux manières :

- **approche de haut en bas** (*top down* en anglais), également appelée **programmation dynamique descendante** ;
- **approche de bas en haut** (*bottom up* en anglais), également appelée **programmation dynamique ascendante**.

1 Approche de haut en bas

L'approche de haut en bas est une évolution naturelle de l'approche récursive dont on conserve la logique, avec la particularité qu'on conserve le résultat des calculs effectués pour éviter d'avoir à les exécuter à plusieurs reprises. Cette approche de programmation dynamique descendante est également appelée **mémoïsation** (parfois orthographiée « mémoïzation » en référence au terme anglais correspondant *"memoization"*).

→ Dans l'approche de haut en bas, on ne s'intéresse pas à l'ordre dans lequel les calculs ont lieu, on évite seulement de les répéter inutilement.

2 Approche de bas en haut

Dans l'approche de bas en haut, on détermine d'abord l'ordre dans lequel les sous-problèmes doivent être traités. On commence par résoudre le premier sous-problème trivial, puis on passe au suivant. On part des sous-problèmes simples et on évolue de manière ascendante pour résoudre le problème global en stockant les résultats intermédiaires. Cette approche de programmation dynamique ascendante est également appelée **tabulation**.

→ Dans les deux cas on effectue les mêmes calculs et on conserve les résultats intermédiaires. On utilise donc de l'espace mémoire pour nous permettre de réduire les temps de calculs et donc de gagner du temps.

Nous allons maintenant appliquer ces deux approches de manière concrète aux calculs des nombres de Fibonacci.

3 | Application à la suite de Fibonacci

a. Approche de haut en bas appliquée à la suite de Fibonacci

Nous allons faire évoluer notre algorithme récursif initial pour disposer d'une structure de données nous permettant de stocker les calculs à mesure qu'ils sont effectués. Ainsi, lors d'un appel pour un calcul, on commencera par vérifier si la valeur a déjà été calculée :

- si c'est le cas, on pourra la retourner directement à partir de la structure de données, sans avoir à lancer de nouveaux calculs ;
- si la valeur n'a pas encore été calculée, les appels correspondants seront lancés comme dans notre implémentation initiale, et on ajoutera le résultat obtenu à notre structure de données afin de pouvoir l'exploiter les fois suivantes.

Nous utilisons un **dictionnaire** pour stocker les résultats des calculs effectués.

```
memo = {}
```

```
def fibonacci(n):
```

```
if n in memo:
    return memo[n]
if n == 0:
    resultat = 0
elif n == 1:
    resultat = 1
else:
    resultat = fibonacci(n-1) + fibonacci(n-2)
memo[n] = resultat
return resultat
```

Lançons le calcul pour $n = 6$:

```
print(fibonacci(6))
# affiche 8
```

Observons maintenant le contenu du dictionnaire de données :

```
print(memo)
# affiche {1: 1, 0: 0, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8}
```

> Le premier élément du dictionnaire est fib(1) et non fib(0) c'est pourquoi il est le premier terme à être calculé (première feuille en bas à gauche de l'arbre de récurrence).

Pour les petites valeurs de n, la différence n'est pas palpable, et les différences au niveau de l'algorithme peuvent paraitre minimes mais le gain de performance est très net : ainsi le calcul pour $n = 50$ qui nécessitait plus d'une heure dans notre implémentation récursive simple est désormais quasiment instantané.

```
print(fibonacci(50))
# affiche (instantanément) la valeur 12586269025
```

Consultons la taille du dictionnaire de données auquel nous avons eu recours :

```
print(len(memo))
# affiche 51
```

À retenir

> La version mémoïsée de notre algorithme nous permet de calculer rapidement les valeurs pour des nombres plus importants. En effet, toute valeur calculée étant stockée dans le dictionnaire, elle peut ensuite être aussitôt retournée sans nouveau calcul.

Nous pouvons désormais calculer des valeurs plus importantes :

```
print(fibonacci(100))
# affiche 354224848179261915075
```

```
print(fibonacci(1000))
# affiche
434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798
```

Toutefois si on augmente les valeurs de n, on finit par atteindre une limite liée au nombre d'appels récursifs.

```
print(fibonacci(10000))
# affiche une erreur : RecursionError: maximum recursion depth exceeded in comparison
```

Nous atteignons ici les limites du nombre d'appels récursifs permis par défaut par l'interpréteur Python. Cette limite, variable selon les configurations, est modifiable si nécessaire.

→ La mémoïsation nous a permis de calculer des valeurs qui auraient été impossibles à obtenir avec la version récursive simple de notre algorithme initial.

Nous avons ainsi réalisé une implémentation manuelle de la mémoïsation, mais cette fonctionnalité est disponible dans la bibliothèque standard de Python, avec le cache LRU de la bibliothèque `functools`.

Rappel

> Le sigle LRU signifie *Least Recently Used*, faisant référence aux valeurs les plus récemment utilisées.

On mobilise ce cache de la manière suivante :

```
from functools import lru_cache
```

Une fois le cache LRU importé, on l'applique à une fonction à l'aide d'un décorateur. Le fonctionnement précis des décorateurs sort du cadre de ce cours mais leur emploi est très simple : on fait simplement précéder la définition de la fonction à décorer d'une ligne indiquant l'emploi du décorateur par le nom de ce dernier précédé d'un signe `@`.

```
@lru_cache()
def fibocache(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibocache(n-1) + fibocache(n-2)
```

Notre fonction `fibocache` dispose ainsi automatiquement d'un cache, sans avoir à déclarer ni à gérer la structure de données correspondante.

```
print(fibocache(50))
# affiche (instantanément) la valeur 12586269025
```

Astuce

> Par défaut le cache LRU a une taille maximale de 128 entrées, mais on peut augmenter cette taille en le précisant sous forme de paramètre optionnel du décorateur ; par exemple : `@lru_cache(maxsize=1000)`.

Nous allons maintenant nous intéresser à l'autre manière possible de programmation dynamique, l'approche de bas en haut.

b. Approche de bas en haut appliquée à la suite de Fibonacci

Dans l'approche de bas en haut, on part des sous-problèmes simples et on évolue de manière ascendante pour résoudre le problème global en stockant les résultats intermédiaires. On effectue donc les mêmes calculs que dans l'approche de haut en bas, mais en prenant en compte l'ordre dans lequel ils sont effectués.

Les valeurs supérieures sont calculées en séquence à partir des valeurs initiales préalablement définies ou calculées.

```
conteneur = {}
def fibomontant(n):
    conteneur[0] = 0
    conteneur[1] = 1
    for i in range(2, n+1):
        resultat = conteneur[i-1] + conteneur[i-2]
        conteneur[i] = resultat
    return resultat

print(fibomontant(50))
# affiche (instantanément) la valeur 12586269025
```

Cet algorithme n'est plus récursif mais itératif. De plus son fonctionnement séquentiel nous permet de le simplifier davantage en se passant totalement du conteneur de données. En effet, il suffit de connaître les deux valeurs précédentes pour pouvoir calculer les suivantes.

La version itérative de Fibonacci peut donc prendre la forme suivante :

```
def fibiteratif(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

Comme pour la version descendante, la version ascendante produit une amélioration très nette par rapport à notre algorithme récursif initial.

```
print(fibiteratif(50))
# affiche (instantanément) la valeur 12586269025
```

Cet algorithme présente en outre l'avantage de ne pas être limité par le nombre d'appels récursifs, puisqu'il n'en effectue aucun.

> Conclusion :
>
> Nous avons dans un premier temps illustré les limites de l'approche récursive avec le calcul des nombres composant la suite de Fibonacci : un ordinateur moderne montre vite ses limites pour des valeurs pourtant relativement faibles.
> La constatation d'un nombre significatif de sous-problèmes se répétant nous a conduit à la présentation d'une approche de programmation dite dynamique, proposant une optimisation de ce type de problème. Nous avons caractérisé ce paradigme de programmation et précisé qu'il était possible de le mettre en œuvre selon deux approches distinctes, de haut en bas ou de bas en haut.
> Nous avons pu constater le gain significatif de performance apporté par cette approche algorithmique sur les nombres de Fibonacci, par l'évitement des répétitions inutiles de mêmes traitements grâce au stockage de leurs résultats.
> Ainsi, ce paradigme de programmation consomme plus de mémoire, au profit du processeur qu'il sollicite nettement moins.