Algorithmique

Programmation dynamique

Cours

Introduction

Contexte historique

La **programmation dynamique** est une méthode algorithmique pour résoudre des problèmes d'optimisation. Le concept a été introduit au début des années 1950 par **Richard Bellman**.

À l'époque, le terme « programmation » signifiait *planification*, et non programmation informatique au sens actuel.

L'idée fondamentale est de décomposer un problème en **sous-problèmes**, de résoudre chaque sous-problème une seule fois, et de **mémoriser les résultats** pour éviter de les recalculer.

Richard Bellman
(1920–1984)

Définition

La **programmation dynamique** consiste à résoudre un problème en le décomposant en **sous-problèmes qui se chevauchent**, en résolvant chaque sous-problème une seule fois et en **stockant les résultats intermédiaires** dans un tableau ou un dictionnaire pour les réutiliser directement.

Elle s'applique lorsqu'un problème vérifie deux propriétés :

1. **Chevauchement de sous-problèmes** : les mêmes sous-problèmes sont résolus plusieurs fois lors d'une approche naïve.
2. **Sous-structure optimale** : la solution optimale du problème se construit à partir des solutions optimales de ses sous-problèmes.

Prog. dynamique vs Diviser pour régner

Dans le paradigme « diviser pour régner » (tri fusion, recherche dichotomique…), les sous-problèmes sont **indépendants** : on ne recalcule jamais la même chose. En programmation dynamique, les sous-problèmes **se chevauchent** : sans mémorisation, on les recalculerait un très grand nombre de fois.

La suite de Fibonacci

Définition

La suite de Fibonacci est définie par la relation de récurrence :

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \quad \text{pour } n \geq 2.$$

Les premiers termes sont : $0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89\ldots$

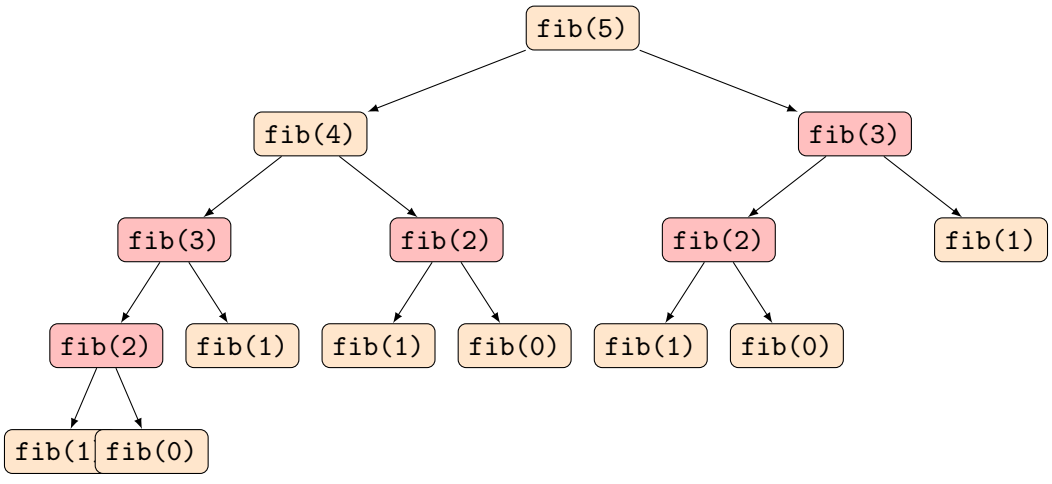
Algorithme récursif naïf

La définition mathématique se traduit directement en Python :

Version naïve

```
def fib(n):
    """Renvoie le n-ième terme de la suite de Fibonacci."""
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Arbre des appels pour `fib(5)` :



`fib(x)` = nœud recalculé inutilement

Des calculs inutiles

`fib(3)` est calculé **2 fois**, `fib(2)` **3 fois**, `fib(1)` **5 fois**. De manière générale, pour calculer `fib(n)`, le nombre d'appels récursifs est de l'ordre de 2^n.

`fib(40)` nécessite ainsi plus de **300 millions d'appels** et prend plusieurs secondes. C'est une complexité **exponentielle** : $O(2^n)$.

Exercice 1 — Observer la lenteur

1. Sans l'exécuter, estimer le nombre d'appels nécessaires pour `fib(6)` en dessinant l'arbre des appels.

   .....

   .....

2. En Python, mesurer le temps d'exécution de `fib(35)` puis de `fib(40)` avec le module `time`. Que remarque-t-on ?

   .....

   .....

   .....

Mémoïsation (approche descendante)

Principe

La **mémoïsation** consiste à mémoriser dans un dictionnaire les résultats déjà calculés. Avant chaque calcul, on vérifie si le résultat est déjà connu. Si oui, on le renvoie directement sans recalculer.

Cette approche est dite **descendante** (*top-down*) : on part du problème général et on descend vers les sous-problèmes, en mémorisant chaque résultat au passage.

Fibonacci avec mémoïsation

```
def fib_memo(n, cache={}):
    """Fibonacci avec mémoïsation (cache dictionnaire)."""
    if n in cache:
        return cache[n]      # résultat déjà calculé → renvoyer directement
    if n <= 1:
        return n
    cache[n] = fib_memo(n - 1, cache) + fib_memo(n - 2, cache)
    return cache[n]
```

Avec la mémoïsation, chaque valeur $F(k)$ pour k de 0 à n n'est calculée **qu'une seule fois** : la complexité devient $O(n)$.

Exercice 2 — Mémoïsation

1. Combien d'appels à `fib_memo` sont nécessaires pour calculer `fib_memo(5)` (en supposant le cache vide au départ) ?

   .....

   .....

2. Comparer le temps d'exécution de `fib_memo(40)` avec celui de `fib(40)`. Quelle complexité temporelle la mémoïsation apporte-t-elle ?

   .....

   .....

Programmation dynamique — approche ascendante

Principe

L'**approche ascendante** (*bottom-up*) consiste à calculer les solutions des **plus petits sous-problèmes en premier** et à les stocker dans un tableau, puis à remonter jusqu'à la solution du problème initial.

Contrairement à la mémoïsation, **aucune récursion** n'est utilisée : on remplit le tableau de gauche à droite avec une simple boucle.

Fibonacci bottom-up

```python
def fib_dp(n):
    """Fibonacci par programmation dynamique (approche ascendante)."""
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
```

Visualisation du tableau `dp` pour `fib_dp(8)` :

i	0	1	2	3	4	5	6	7	8
dp[i]	0	1	1	2	3	5	8	13	**21**

Optimisation de la mémoire

Pour Fibonacci, on n'a besoin que des **deux dernières valeurs** pour calculer la suivante. On peut se passer du tableau et utiliser deux simples variables, ramenant la complexité en espace de $O(n)$ à $O(1)$:

Version optimisée — O(1) espace

```
def fib_opt(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

Application : le rendu de monnaie

Échec de l'algorithme glouton

L'**algorithme glouton** pour le rendu de monnaie consiste à choisir à chaque étape la plus grande pièce possible jusqu'à atteindre le montant voulu.

Exercice 3 — L'algorithme glouton ne fonctionne pas toujours

1. **Système euro** — pièces disponibles : 50, 20, 10, 5, 2, 1 . Appliquer l'algorithme glouton pour rendre **48** . Combien faut-il de pièces ?

   . . . . . .

   . . . . . .

2. **Système impérial fictif** — pièces disponibles : 30, 24, 12, 6, 3, 1 £. Appliquer l'algorithme glouton pour rendre **48 £**.

   . . . . . .

   . . . . . .

3. Trouver une solution utilisant **moins** de pièces. Conclure.

   . . . . . .

   . . . . . .

Solution par programmation dynamique

Relation de récurrence

On construit un tableau `dp` de taille $n+1$ (pour un montant n) tel que `dp[i]` est le **nombre minimum de pièces** pour rendre le montant i.

Initialisation : `dp[0] = 0` (rendre 0 nécessite 0 pièce).

Récurrence : pour chaque montant i de 1 à n, et pour chaque pièce c telle que $c \leq i$:

$$\texttt{dp}[i] = \min\big(\texttt{dp}[i],\ \texttt{dp}[i-c]+1\big)$$

Rendu de monnaie — programmation dynamique

```python
def rendu_monnaie(montant, pieces):
    """
    Renvoie le nombre minimum de pièces pour rendre 'montant'.
    montant : int, montant à rendre
    pieces  : list, valeurs des pièces disponibles
    """
    dp = [float('inf')] * (montant + 1)
    dp[0] = 0
    for i in range(1, montant + 1):
        for c in pieces:
            if c <= i and dp[i - c] + 1 < dp[i]:
                dp[i] = dp[i - c] + 1
    return dp[montant]
```