Chapitre D.5 - La programmation dynamique

I. Définition

La programmation dynamique est une technique de programmation consistant à faire en sorte de ne pas calculer plusieurs fois la même chose, en stockant des résultats intermédiaires dans une structure de données.
Tout comme la méthode diviser pour régner, il s'agit de construire la solution du problème en divisant celui-ci en sous-problème plus petit. On utilise la programmation dynamique lorsque les sous-problème ne sont pas indépendant (on dit qu'ils se chevauchent). Contrairement à la méthode diviser pour régner, on stocke les résultats intermédiaires de façon à les réutiliser ultérieurement pour résoudre un sous-problème identique à un sous-problème déjà résolu.

II. La mémoïsation

En programmation dynamique, on conserve les résultats des sous-problèmes dans une structure de données. Ainsi, chaque sous-problème identique n'est calculer qu'une unique fois. On ne doit pas le recalculer dans le cadre d'un prochain appel récursif.
Ce mécanisme est appelé la mémoïsation.

Exemple :

Dans un système de pièces non canonique, l'algorithme glouton ne donne pas toujours la solution optimale.
Exemple de pièces disponibles : {1, 3, 4} centimes
Montant à rendre : 6 centimes
Glouton (prend la plus grande possible en premier) :

- 4 (reste 2)
- 1 (reste 1)
- 1 (reste 0)
- Total : 3 pièces

Optimale (par programmation dynamique) :

- 3 (reste 3)
- 3 (reste 0)
- Total : 2 pièces (meilleure solution)

A. Bottom-Up

Dans cette approche, on commence par résoudre les petits sous-problèmes, puis on utilise leurs solutions pour résoudre progressivement les sous-problèmes plus grands jusqu'à atteindre le problème final.

- Avantage : On évite la récursion et l'explosion d'appels récursifs, ce qui optimise la mémoire et la rapidité.
- Inconvénient : On doit toujours remplir toute la table, même si certaines valeurs ne sont jamais utilisées.

```python
def rendu_monnaie_non_canonique(montant, pieces):
    # Initialisation : tableau des solutions, avec une valeur infinie par défaut
    dp = [float('inf')] * (montant + 1)
    dp[0] = 0  # Rendre 0 centime nécessite 0 pièce

    # Remplissage du tableau dp
    for i in range(..., montant + 1):
        for piece in pieces:
            if i >= piece:
                dp[i] = min(dp[i], ...)

    # Reconstruction de la solution
    if dp[montant] == float('inf'):
        return None  # Impossible de rendre la monnaie
    return dp[montant]


pieces = [1, 3, 4]
montant = 6
print("Nombre minimal de pièces :", rendu_monnaie_non_canonique(montant, pieces))
```

B. Top-down

Dans cette approche, on part du problème principal et on le décompose en sous-problèmes, qu'on résout récursivement avec une mémoïsation pour éviter les calculs redondants.
On définit une fonction récursive qui :
Cherche la meilleure combinaison pour montant en appelant récursivement la fonction pour montant - piece.
Stocke les résultats déjà calculés pour éviter de refaire les mêmes calculs (mémoïsation).

```python
def rendu_monnaie_top_down(montant, pieces, memo={}):
    if montant == 0:
        return 0  # Pas besoin de pièce pour rendre 0
    if montant in memo:
        return ...  # Retourne le résultat déjà calculé

    for piece in pieces:
        if montant >= piece:
            min_pieces = min(min_pieces, 1 + ...)

    memo[montant] = ...
    return min_pieces

pieces = [1, 3, 4]
montant = 6
print("Nombre minimal de pièces (top-down) :", rendu_monnaie_top_down(montant, pieces))
```

- Avantage : On ne calcule que les valeurs nécessaires, ce qui peut être plus efficace pour certains problèmes.
- Inconvénient : Utilise la récursion, ce qui peut entraîner une consommation mémoire plus importante (pile d'appels récursifs).

III. Choisir la bonne méthode algorithmique

- 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 (qui ne se chevauchent pas), chaque sous-problème, et combine les solutions des sous-problèmes pour former une solution du problème initial.
- Programmation dynamique : divise un problème en sous-problèmes qui sont non indépendants (qui se chevauchent), et cherche (et stocke) des solutions de sous-problèmes de plus en plus grands