REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'enseignement supérieur et de la recherche scientifique
Université Larbi Ben M'Hidi Oum El Bouaghi



Cours Méthodes algorithmiques

1ere année master AD

Chapitre 3
La programmation dynamique









Dr.Belhouchette

Introduction à la programmation dynamique

- La programmation dynamique est une méthode algorithmique qui facilite la résolution de problèmes complexes en les divisant en sous-problèmes plus petits, qui sont mémorisés pour éviter les calculs redondants.

- Cette approche est idéale lorsque les sous-problèmes se répètent souvent dans le processus de résolution, ce qui permet de gagner en efficacité.

- Contrairement aux méthodes gloutonnes, où chaque étape suit la meilleure solution immédiate sans garantie de solution optimale, la programmation dynamique repose sur la réutilisation de solutions antérieures pour construire une solution globale optimale.

Comparaison avec d'autres Méthodes

La programmation dynamique se distingue des autres techniques par sa capacité à gérer les répétitions de sous-problèmes. Par exemple :

- Les méthodes gloutonnes ne visent qu'à optimiser chaque décision localement, sans considérer l'ensemble.

- La division et conquête comme pour le tri fusion), divise un problème en sous-parties indépendantes. Mais, en programmation dynamique, les sous-problèmes ne sont pas indépendants et leur solution peut se répéter plusieurs fois.

- Le principe de **mémorisation** est crucial en programmation dynamique, économisant du temps et des ressources en stockant chaque solution intermédiaire pour éviter de recalculer.

Conditions d'Utilisation

Les critères ou propriétés spécifiques qu'un problème doit satisfaire pour que la programmation dynamique soit une technique appropriée pour le résoudre efficacement :



Conditions d'Utilisation

1. **Sous-structure optimale**

Un problème a une sous-structure optimale si sa solution optimale globale peut être construite à partir des solutions optimales de ses sous-problèmes.

Par exemple dans le problème du plus court chemin, la distance minimale entre deux points est constituée des distances minimales entre les points intermédiaires.

Cela permet de combiner des sous-solutions optimales pour obtenir la solution optimale du problème global.

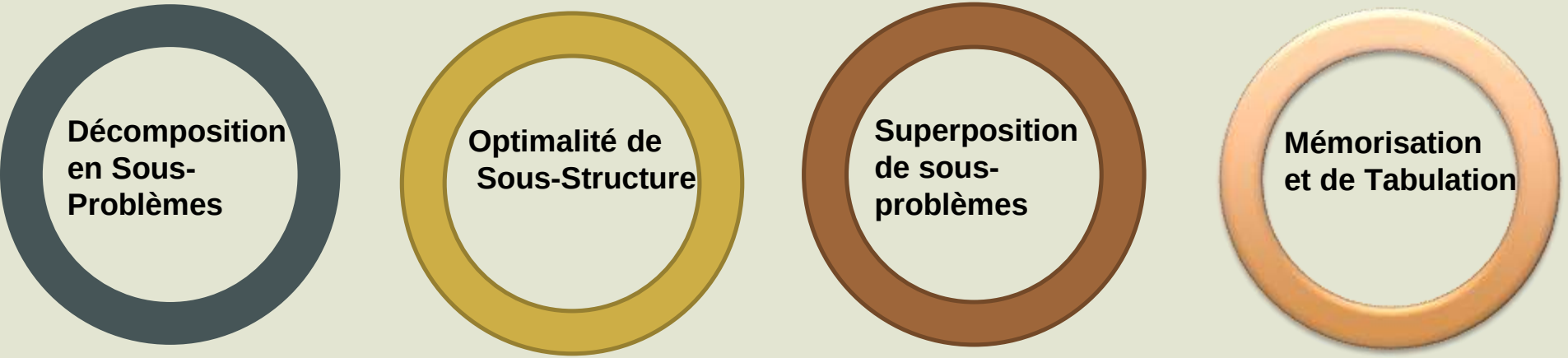
Conditions d'Utilisation

2. Chevauchement des sous-problèmes

Cette condition signifie que les mêmes sous-problèmes apparaissent plusieurs fois dans le processus de résolution du problème global.

Par exemple dans le calcul des nombres de Fibonacci, chaque nombre est calculé plusieurs fois de manière répétitive dans une approche récursive classique. Grâce au chevauchement, on peut mémoriser les valeurs intermédiaires pour éviter de refaire les mêmes calculs, ce qui rend la programmation dynamique très efficace ici.

Principes Fondamentaux de la Programmation Dynamique



Principes Fondamentaux de la Programmation Dynamique

1. Décomposition en Sous-Problèmes

- La programmation dynamique commence par la décomposition d'un problème complexe en morceaux plus petits et plus gérables.

- Chaque sous-problème représente une partie du problème original, et résoudre ces sous-problèmes de manière efficace facilite l'élaboration d'une solution pour le problème principal.

- Par exemple, dans le cadre du problème du sac à dos, l’idée est de décider pour chaque objet s'il doit être inclus ou non, ce qui entraîne une série de décisions plus simples à traiter.

Principes Fondamentaux de la Programmation Dynamique

2. Optimalité de Sous-Structure

- Un problème présente une sous-structure optimale lorsque la solution optimale du problème global peut être construite à partir des solutions optimales de ses sous-problèmes.

- En d'autres termes, la meilleure solution pour le problème complet repose sur les meilleures solutions trouvées pour ses éléments constitutifs.

- Cela se voit dans des applications telles que la recherche du chemin le plus court dans un graphe, où la distance entre deux nœuds peut être déterminée à partir des distances entre les nœuds adjacents.

Principes Fondamentaux de la Programmation Dynamique

3. Superposition de sous-problèmes

- Ce principe stipule que de nombreux sous-problèmes sont souvent identiques et peuvent apparaître à plusieurs reprises lors de la résolution du problème global.

- Les mêmes calculs peuvent être effectués à divers endroits, rendant la mémorisation des résultats intermédiaires particulièrement avantageuse.

Principes Fondamentaux de la Programmation Dynamique

3. Superposition de sous-problèmes

Prenons l'exemple de la suite de Fibonacci :

En utilisant une approche récursive naïve, on peut calculer F(n) en fonction de F(n−1) et F(n−2). En résolvant de manière **récursive**, nous calculons F(n−1)et F(n−2) plusieurs fois.

En utilisant la **mémorisation**, nous stockons les résultats de F(n−1) et F(n−2) pour les réutiliser, évitant ainsi les recalculs.

Principes Fondamentaux de la Programmation Dynamique

4. Techniques de Mémorisation et de Tabulation

Les deux principales approches de programmation dynamique sont:

la **mémorisation** (approche descendante, ou "top-down")

la **tabulation** (approche ascendante, ou "bottom-up").

Approches de la Programmation Dynamique

Mémorisation (Top-Down)

La **mémorisation** consiste à utiliser une approche descendante pour résoudre un problème en décomposant récursivement le problème en sous-problèmes, tout en sauvegardant les résultats de chaque sous-problème pour éviter les calculs redondants.

Approches de la Programmation Dynamique



Décomposition en Sous-Problèmes

Le problème initial est divisé en plusieurs sous-problèmes plus petits. Chaque sous-problème est résolu de manière récursive.

Stockage des Résultats

À chaque résolution d'un sous-problème, le résultat est sauvegardé dans une structure de données (comme un dictionnaire ou un tableau).

Vérification

Avant de résoudre un sous-problème, l'algorithme vérifie si le résultat est déjà stocké. Si oui, il retourne ce résultat directement, sans recalculer.

Approches de la Programmation Dynamique

Mémorisation (Top-Down)
Eemple

Dans la série de Fibonacci, chaque nombre est la somme des deux nombres précédents. En utilisant la **mémorisation**, on peut éviter de recalculer plusieurs fois les mêmes valeurs.

```
def fibonacci_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        memo[n] = n
        return n
    memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)
    return memo[n]
```

Exemple illustarifs de la programmationDynamique

Tabulation (Bottom-Up)

La **tabulation** est une approche ascendante qui résout le problème en construisant les solutions de tous les sous-problèmes jusqu'au problème complet, en commençant par les cas de base. Contrairement à la mémorisation, elle n'utilise pas la récursivité et construit la solution de manière itérative.

Approches de la Programmation Dynamique



Cas de base

Les résultats des cas de base sont directement initialisés dans une structure de données (souvent un tableau).

Construction itérative

À partir des cas de base, l'algorithme remplit progressivement le tableau jusqu'à atteindre la solution du problème initial.

Approches de la Programmation Dynamique

Tabulation (Bottom-Up)
Exemple

- Pour calculer le n-ième nombre de Fibonacci avec une approche ascendante, on commence par les valeurs de base et on remplit un tableau jusqu'à la valeur de F(n).

```
def fibonacci_tab(n):
    if n <= 1:
        return n
    fib_table = [0] * (n + 1)
    fib_table[1] = 1
    for i in range(2, n + 1):
        fib_table[i] = fib_table[i - 1] + fib_table[i - 2]
    return fib_table[n]
```

Rendu de monnaie

Cas où l'algorithme échoue :
Considérons un ensemble de pièces C={1,3,4} et une somme à rendre de S=6.
En suivant l'algorithme glouton :

- On commence par utiliser la pièce de 4, ce qui laisse 6−4=2.
- Puis on utilise 2 pièces de 1 pour rendre les 2 unités restantes.
- Résultat : 1 pièce de 4 et 2 pièces de 1, donc un total de 3 pièces.

Cependant, la solution optimale serait d'utiliser 2 pièces de 3, soit un total de 2 pièces, ce qui montre que l'algorithme glouton n'est pas optimal dans ce cas.

Rendu de monnaie

```
def piece_change(pieces, S):
tab = [float('inf')] * (S + 1)

tab[0] = 0

for i in range(1, S + 1):
        for piece in pieces:
            if i >= piece:
tab[i] = min(tab[i], tab[i - piece] + 1)

return tab[S] if tab[S] != float('inf') else -1
```