version pdf

# Synthèse du cours Programmation dynamique

## Sous-structure optimale

**"Programmation dynamique et décomposition en sous-problèmes"**

La **programmation dynamique** est une méthode de conception d'algorithmes qui s'applique à des *problèmes d'optimisation* où la solution finale doit rendre maximale ou minimale une certaine fonction *objectif*. Un algorithme de programmation dynamique n'est pas une heuristique comme c'est souvent le cas avec les algorithmes gloutons, il est en général moins performant qu'un algorithme glouton, mais il détermine une solution exacte, pas une solution approchée.

**"Exemples où s'applique la programmation dynamique"**

| Problème | Entrées | Nature de la solution | Fonction objectif à optimiser |
|---|---|---|---|
| Rendu de monnaie | Un montant à rendre | Une liste de pièces de somme égale au montant | Le nombre de pièces doit être minimal |
| Sac à dos | Listes de poids et de valeurs d'objets et une capacité de sac à dos | Une sélection d'objets compatible avec la capacité du sac | La somme des valeurs des objets doit être maximale |
| Alignement de séquences | Deux séquences de gènes | Un alignement des deux séquences avec éventuellements des trous | La somme des pénalités doit être minimale, sachant que chaque trou ou différence dans |

| Problème | Entrées | Nature de la solution | Fonction objectif à optimiser |
|---|---|---|---|
| | | | l'alignement est pénalisé |
| Chemin minimal dans un triangle de nombres | Triangle de nombres | Un chemin du sommet vers la base du triangle | La somme des nombres sur le chemin doit être minimale |

Voici les trois étapes de la **programmation dynamique** :

- **Étape 1 *Décomposition en sous-problèmes* :** On exprime la solution optimale du problème en fonction de solutions optimales de *sous-problèmes* : on fait apparaître une **sous-structure optimale** à travers une **relation de récurrence**.
- **Étape 2 *Calcul et mémorisation des solutions des sous-problèmes* :** On résout les *sous-problèmes* en partant du *bas* : les cas de base et en remontant vers le *haut* : le problème initial, et on **mémorise les résultats intermédiaires dans une structure de données** (par exemple un tableau).
- **Étape 3 *Construction de la solution finale* :** À l'issue de l'étape 2, on a l'optimum de la fonction *objectif* mais il reste à **construire la solution finale**. On peut le faire en parcourant dans l'autre sens, du *haut* vers le *bas*, la structure de données où on a mémorisé les résultats de tous les *sous-problèmes*.

# Mémoriser les solutions des sous-problèmes pour ne pas les recalculer

**"Programmation dynamique et redondance des calculs"**

La **programmation dynamique** détermine une solution optimale d'un problème à partir des solutions optimales de sous-problèmes (**sous-struture optimale**). Cela nous conduit naturellement à l'écriture d'un *algorithme récursif* de résolution.

Cependant, contrairement aux algorithmes **Diviser Pour Régner**, où les sous-problèmes sont indépendants, dans les problèmes de **programmation dynamique** les sous-problèmes peuvent se chevaucher.

Un algorithme *récursif* naïf va donc calculer plusieurs fois les solutions des mêmes sous-problèmes. La **mémoïsation** permet d'éviter ces redondances de calcul : on améliore notablement la *complexité temporelle* (d'exponentielle à linéaire pour le rendu de monnaie) au prix d'une plus grande *compelxité spatiale* :

- on enregistre chaque solution de sous-problème dans une structure de données `memo` (dictionnaire ou tableau avec accès en temps constant)
- l'algorithme récursif vérifie d'abord si le sous-problème traité n'est pas déjà stocké dans `memo` avant de calculer sa solution et enregistre toute nouvelle solution dans `memo`

En **programmation dynamique**, un algorithme *récursif* procède de *haut* en *bas* en décomposant d'abord le problème et il doit être *mémoisé* pour être efficace. Mais, avec un algorithme *itératif*, on peut aussi effectuer les calculs de *bas* en *haut* en progressant des plus petits sous-problèmes jusqu'aux plus grands et en enregistrant toutes les solutions calculées dans un tableau.

Dans tous les cas, l'objectif de la **programmation dynamique** est d'éviter *la redondance des calculs* : on calcule toutes les solutions de sous-problèmes nécessaires mais une seule fois !

> **"Différences entre *Diviser Pour Régner* et *Programmation dynamique*"**
>
> Les méthodes de **programmation dynamique** et **Diviser Pour Régner** permettent de résoudre un problème en le décomposant en sous-problèmes mais se distinguent en de nombreux points :
>
> - Dans la méthode **Diviser Pour Régner** les sous-problèmes sont indépendants et ne se chevauchent pas, alors qu'en **programmation dynamique** cette contrainte n'existe pas et les sous-problèmes peuvent se chevaucher.
> - La méthode **Diviser Pour Régner** est choisie principalement avec l'objectif de réduire la *complexité temporelle*, alors qu'en **programmation dynamique** l'objectif principal est de construire une solution *correcte*.
> - En général, avec **Diviser Pour Régner** la taille des sous-problèmes est une fraction de celle du problème initial, alors qu'en **programmation dynamique** elle peut être simplement décrémentée de quelques unités.
>
> En résumé, la **programmation dynamique** s'applique à une plus grande variété de problèmes. **Diviser Pour Régner** peut être vue comme un cas particulier de la

> **programmation dynamique** où la décomposition en sous-problèmes est toujours la même et où les sous-problèmes sont indépendants.

# Exemple du rendu de monnaie

> **"Spécification du problème"**
>
> On se place dans la position du caissier qui doit rendre en monnaie un certain montant avec un nombre minimal de pièces. On suppose que le caissier dispose en nombre illimité de toutes les valeurs de pièces disponibles. L'ensemble des valeurs de pièces disponibles constitue le *système monétaire*.
>
> Il s'agit d'un **problème d'optimisation** dont la spécification est la suivante :
>
> - **Entrée du problème :** un montant à rendre et une liste de valeurs de pièces d'un système monétaire ; on suppose qu'on dispose d'un nombre illimité de pièces de chaque valeur et qu'on dispose de pièces de 1 ainsi on peut toujours rendre la monnaie
> - **Sortie du problème :** une liste de pièces dont la somme est égale au montant à rendre et dont le nombre de pièces est minimal

> **"Version récursive avec mémoïsation dans un dictionnaire"**

```python
def rendu_monnaie_memo(montant, pieces):
    """Renvoie le nombre minimal de pièces pour rendre la monnaie
    sur montant avec le système monétaire pieces qui contient une pièce de 1"""

    def rendu_monnaie(montant, pieces):
        if montant not in memo:
            rmin = montant # montant pièces de 1, pire des cas
            for p in pieces:
                if p <= montant:
                    rmin = min(rmin, 1 + rendu_monnaie(montant - p, pieces))
            memo[montant] = rmin
        return memo[montant]

    memo = {0: 0}
    return rendu_monnaie(montant, pieces)

assert rendu_monnaie_memo(8, [1, 4, 6]) == 2
```

## "Version itérative avec mémorisation dans un tableau"

```python
def rendu_monnaie_dyna(montant, pieces):
    """Renvoie le nombre minimal de pièces pour rendre la monnaie
    sur montant avec le système monétaire pieces qui contient une pièce de 1
    """
    memo = [0 for _ in range(montant + 1)]
    for m in range(1, montant + 1):
        memo[m] = montant # m pièces de 1
        for p in pieces:
            if p <= m:
                memo[m] = min(memo[m], 1 + memo[m - p])
    return memo[montant]
```

## "Version itérative avec construction d'une liste solution"

```python
def rendu_monnaie_dyna_solution(montant, pieces):
    """Renvoie un couple avec :
    - le nombre minimal de pièces pour rendre la monnaie
    sur montant avec le système monétaire pieces qui contient une pièce de 1
    - une liste de pièces réalisant cet optimum
    """
    memo = [0 for _ in range(montant + 1)]
    choix = [0 for _ in range(montant + 1)]
    for m in range(1, montant + 1):
        memo[m] = montant # m pièces de 1
        for p in pieces:
            if p <= m:
                if memo[m - p] < memo[m]:
                    memo[m] = 1 +  memo[m - p]
                    choix[m] = p
    solution = [choix[montant]]
    m = montant - choix[montant]
    while m != 0:
        solution.append(choix[m])
        m = m - choix[m]
    return (memo[montant], solution)

assert rendu_monnaie_dyna_solution(8, [1, 4, 6]) == (2, [4, 4])
```