Université Clermont Auvergne
ISIMA FISA 1
Algorithmique

Florent Foucaud
2025-2026

TP - Algorithmes de programmation dynamique

La *programmation dynamique* est une technique algorithmique qui consiste à calculer la solution pour des sous-problèmes, et où la solution du sous-problème suivant dépend directement de un ou plusieurs sous-problèmes précédemment calculés.

Souvent, c'est le moyen de rendre itératif un algorithme récursif, en passant d'une complexité exponentielle à une complexité polynomiale.

Les sous-problèmes sont souvent stockés dans une tableau, ou bien, on peut aussi faire de la programmation dynamique sur des structures arborescentes ou des graphes orientés acycliques. Dans ce TP ce sera sur des tableaux ou listes.

Exercice 1 (Rendu de monnaie)**.**

On a une liste de dénominations de pièces et billets, par exemple `L=[1,2,5,10,20,50,100,200,500]`. Étant donné un entier `n` de monnaie à rendre en euros, on veut savoir quelle est la meilleure combinaison de pièces/billets (on souhaite minimiser le nombre total de pièces/billets à rendre).

On pourrait proposer un algorithme glouton pour ce problème (parfois fait en cours d'Algo en BUT1) : on choisit la plus grande dénomination `d` qui ne dépasse pas `n`, et on continue avec `n-d` jusqu'à arriver à 0. Cependant, cet algorithme n'est pas optimal : par exemple pour `L=[5,10,20,25]` et `n=40`, cet algorithme va renvoyer 3 (25+10+5) alors que l'optimum est 2 (20+20). Il faut donc trouver une méthode plus exhaustive...

1. On peut résoudre le problème récursivement : si `n` est dans la liste `L`, on renvoie 1 ; sinon, on renvoie le minimum de toutes les valeurs obtenues récursivement pour `n-d`, où `d` est un élément de `L` qui représente la *dernière* pièce/billet rendue.

   Coder une solution récursive `MonnaieRec(n,L)` et tester (on s'intéresse tout d'abord au nombre optimal de pièces/billets, pas forcément à la combinaison exacte : par exemple pour `n=12` la solution est `2` : un billet de 10 et une pièce de 2).

   Jusqu'à quelle valeur cet algorithme est-il capable de calculer la solution rapidement (<30sec) ? Quelle est sa complexité ?

   Vous pouvez tester avec 36 et la liste donnée ci-dessus et trouver 4. Il est aussi intéressant de tester avec $n=10$ et `L=[1,5,7]` et de bien trouver 2.

2. Pour éviter de recalculer des valeurs un grand nombre de fois, on peut stocker les valeurs optimales pour tout `n` dans une liste `valeurs`, et remplir la liste de l'indice 1 à `n` : `valeurs[i]` contiendra la valeur optimale pour `i` euros. Cette technique est appelée *programmation dynamique*.

   Coder un tel algorithme `MonnaieProgDyn(n,L)`. Quelle est sa complexité ?

3. Modifier l'algorithme pour qu'il affiche également une solution (une combinaison optimale de pièces/billets). La solution pourra être stockée comme un dictionnaire dont les clés sont les éléments de `L` et les valeurs, le nombre de pièces/billets de chaque dénomination. Pour cela, il faut "backtracker" : partir de la valeur de la solution finale et déterminer pas à pas, à l'envers, quelles étapes ont été utilisées pour y arriver.

Exercice 2 (Sac à dos)**.**

On dispose d'un container avec une capacité (volume) limitée (CAPACITE), et on veut charger des objets dans le container. Chaque objet a un volume et une utilité donnés. Ceux-ci sont listés comme des couples (volume,utilité). Les volumes sont en m^3, et l'utilité est un entier (qui représente par exemple des milliers d'euros, ou des milliers de vies sauvées). Par exemple :

`A=[(1,3),(2,4),(4,5),(8,8),(9,10),(6,6),(12,15)]`

Ici, le premier objet a un volume de $1m^3$, et et une utilité de 3. Chaque objet ne peut être choisi qu'une seule fois au maximum.

On définit un tableau à 2 dimensions T tel que `T[i][j]` contient la meilleure utilité totale pour un volume total d'au plus `j`, qui utilise des objets parmi les `i` premiers objets de `A`.

Par exemple, `T[0][j]=0` pour tout `j` (car on veut une solution qui utilise 0 objets) et `T[i][0]=0` pour tout `i` (car le volume total est de 0). L'utilité de la solution optimale se retrouvera dans la valeur de `T[len(A)][CAPACITE]` puisqu'on aura droit au budget total et à tous les objets.

1. Cette fois-ci , vous ne vous faites plus avoir : hors de question d'implémenter la solution récursive exponentielle ! Codez une fonction `SacADos(A,budget)` utilisant la programmation dynamique. Elle permet d'optimiser la sélection des objets en fonction du `budget` et de la liste `A` des objets. Dans un premier temps, on s'intéresse seulement à calculer l'utilité totale maximale, pas à la liste des objets à sélectionner pour l'atteindre.

   Pour calculer `T[i][j]`, on va prendre le maximum de deux valeurs :

   - `T[i-1][j]`, qui représente le fait de ne pas utiliser le `i`ème objet,
   - `T[i-1][j-c]+u`, seulement si le coût `c=A[i-1][0]` de l'objet numéro `i` est au plus `j`, et où `u=A[i-1][1]` est l'utilité de l'objet numéro `i` ; cela revient à choisir le `i`ème objet (les indices sont décalés car le premier objet a pour indice 0 dans `A`).

2. Modifier le code pour qu'il affiche une solution optimale (pas seulement la valeur de l'utilité maximum, mais aussi les objets à sélectionner), son coût total et son utilité totale. Il faut, comme à l'exercice précédent, "backtracker".