Université Clermont Auvergne
BUT info 3A
Qualité algorithmique

Florent Foucaud, Pascal Lafourcade et Malika More
2024-2025

TP5 - 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 plutôt, des listes Python).

Vous avez déjà rencontré des algorithmes de programmation dynamique : l'algorithme de Floy-Warshall qui calcule les distances dans un graphe (par calcul de tableau en 2D successifs), vu en BUT1. L'algorithme itératif pour calculer les nombres de Fibonacci du TP1 peut aussi être vu comme un algorithme simple de programmation dynamique.

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 ?

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 (Organisation humanitaire)**.**

L'ONG humanitaire *SaveHumanity* souhaite maximiser son impact positif. Elle dispose d'un budget limité, ainsi que diverses possibilités d'action. Chaque action a un coût et une utilité (nombre de vies sauvées) donnés. Ceux-ci sont listées comme des couples (coût,utilité). Les coûts sont en millions d'euros, et l'utilité en milliers de vies sauvées. Par exemple :

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

Ici, la première action coûte 1 million d'euros, et permet de sauver 3000 vies humaines. Chaque action ne peut être effectuée qu'une seule fois au maximum.

1. Cette fois-ci , vous ne vous faites plus avoir : hors de question d'implémenter la solution récursive exponentielle! Codez une fonction `SaveHumanity(A,budget)` utilisant la programmation dynamique. Elle permet d'optimiser la politique de l'ONG en fonction de son `budget` et de la liste `A` des actions possibles. Dans un premier temps, on s'intéresse seulement au nombre de vies sauvées (utilité totale), pas à la liste des actions à mener.

   Pour cela nous allons remplir un tableau `T` à deux dimensions (en pratique, une liste de listes) tel que `T[i][j]` contient la meilleure utilité totale pour un budget d'au plus `j`, qui utilise des actions parmi les `i` premières actions de la liste `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 budget est de 0). L'utilité de la solution optimale se retrouvera dans la valeur de `T[len(A)][budget]` puisqu'on aura droit au budget total et à toutes les actions.

   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 la `i`ème action,
   - `T[i-1][j-c]+u`, seulement si le coût `c=A[i-1][0]` de l'action numéro `i` est au plus `j`, et où `u=A[i-1][1]` est l'utilité de l'action numéro `i` ; cela revient à choisir la `i`ème action (les indices sont décalés car la première action 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 actions que doit faire l'ONG pour l'atteindre), son coût total et son utilité totale. Il faut, comme à l'exercice précédent, "backtracker".