

TD3 - 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.

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 : 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...

On note $s[i]$ la solution pour la somme i (plus petit nombre de pièces/billets dont la somme est i).

1. Exprimer $s[i]$ selon deux cas : (1) i est dans la liste L des dénominations, ou sinon (2), en fonction de $s[i-d]$ pour tous les d dans L et $d < i$.
2. En déduire un algorithme récursif `MonnaieRec(n,L)` pour calculer la valeur optimale.
3. Estimer de façon grossière la complexité de l'algorithme récursif en fonction de n et de la taille $|L|$ de L en analysant l'arbre des appels récursifs.
4. Le transformer en algorithme itératif `MonnaieProgDyn(n,L)` utilisant le principe de programmation dynamique.
5. Quelle est sa complexité approximative en temps en fonction de n et de la taille $|L|$ de L ? Comme $N = n + |L|$ est la taille de l'entrée, quelle est la complexité en fonction de N ?
6. Quelle modification peut-on faire pour aussi calculer la combinaison optimale (pas juste la valeur)?

Mise en pratique.

1. Coder une première 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 ($<30\text{sec}$)? Vous pouvez tester avec 36 et la liste $L = [1, 2, 5, 10, 20, 50, 100, 200, 500]$ et trouver 4. Il est aussi intéressant de tester avec $n = 10$ et $L = [1, 5, 7]$ et de bien trouver 2.

2. Coder la version itérative `MonnaieProgDyn(n,L)` utilisant la programmation dynamique. Tester sur de grandes valeurs de n .
3. Modifier l'algorithme pour qu'il affiche également une solution (une combinaison optimale de pièces/billets). La solution pourra (par exemple) 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 (Répartition de médicaments).

Une ONG dispose d'un stock de denrées (médicaments) en quantité n . Quand l'ONG distribue une quantité i de denrées à un endroit, l'expérience montre que cela sauve $S[i]$ personnes. Le tableau S a été établi par des experts et mis à disposition à l'ONG. On cherche à calculer le nombre maximal de personnes que l'on pourra sauver avec le stock en le découpant en sous-parties. Par exemple, avec le tableau $S = [1, 3, 2]$ (c'est-à-dire $S[1] = 1$, $S[2] = 3$, $S[3] = 2$) et $n = 3$, on peut sauver :

- soit $1 + 1 + 1 = 3$ personnes en coupant le stock en 3 parties de taille 1 ;
 - ou bien, 2 personnes en laissant le stock en une seule partie de taille 3 ;
 - ou bien, $1 + 3 = 4$ personnes en coupant le stock en 1 partie de taille 1 et 1 partie de taille 2.
1. On note $m[i]$ la meilleure solution (nombre de personnes sauvées) pour un stock de taille i . Exprimer $m[i]$ récursivement, en fonction des valeurs $m[j]$ avec $j < i$, en utilisant aussi S lorsque nécessaire.
 2. En déduire un algorithme récursif **RepartitionRec**(n, S) pour calculer la valeur optimale pour une valeur de n et un tableau $S[1..n]$ donné.
 3. Quelle est la classe de complexité (approximative) en temps de cet algorithme ?
 4. Donner un algorithme itératif **RepartitionProgDyn**(n, S) en utilisant le principe de la programmation dynamique.
 5. Quelle est sa complexité asymptotique en temps ?

Exercice 3 (Problème du sac à dos (knapsack)).

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 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 $T[\text{len}(A)][\text{CAPACITE}]$ puisqu'on aura droit au budget total et à tous les objets.

1. Donner une formule récursive permettant de calculer $T[i][j]$. Pour cela, considérer deux cas : soit on ne sélectionne pas l'objet i et on se ramène à $T[i-1][j]$, soit on le sélectionne et on se ramène aussi à un cas précédent (lequel ?).
2. Écrire l'algorithme de programmation dynamique basé sur cette formule.
3. Quelle est sa complexité asymptotique ?

Mise en pratique.

1. Codez une fonction **SacADos**(A, budget) utilisant la programmation dynamique. Tester avec $A = [(1, 3), (2, 4), (4, 5), (8, 8), (9, 10), (6, 6), (12, 15)]$.
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 "backtracker".

Exercice 4 (Plus grand sous-tableau croissant).

Étant donné un tableau de N entiers, on cherche le plus grand sous-tableau d'éléments croissants (pas forcément consécutifs). Par exemple, pour le tableau $[1, 5, 3, 1, 8]$ des sous-tableaux croissants sont (entre autres) $[1, 8]$, $[1, 5, 8]$, ou encore $[1, 3, 8]$. La taille maximum d'un sous-tableau croissant de TAB est donc 3.

Pour le résoudre on va déterminer la solution optimale de façon incrémentale.

On suppose qu'on a un tableau d'entiers $TAB[1...N]$. Soit $d[i]$ la taille d'un plus long sous-tableau de TAB croissant dont le dernier élément est $TAB[i]$.

1. Exprimer $d[i]$ en fonction des valeurs $d[j]$ avec $j < i$.
2. En déduire un algorithme récursif $STCRec(TAB, N)$ pour calculer *la taille* d'un plus grand sous-tableau de TAB croissant.
3. Quelle est sa complexité asymptotique en fonction de N ?
4. Donner un algorithme itératif $STCProgDyn(TAB, N)$ utilisant la technique de la programmation dynamique.
5. Quelle est sa complexité en temps ?
6. Quelle modification peut-on faire pour aussi calculer le plus long sous-tableau croissant (pas juste sa taille) ?