DIU « Enseigner l'Informatique au Lycee » - UE 5

Programmation dynamique

Nous avons vu pour la fonction de calcul de la somme maximale des valeurs d'une branche d'un arbre que la programmation dynamique permet de trouver une solution optimale là où un algorithme glouton ne la trouve pas forcément. La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires et en suivant une règle d'optimalité.

La méthode de programmation dynamique, comme la méthode diviser-pour-régner, résout des problèmes en combinant des solutions de sous-problèmes. Les algorithmes diviser-pour-régner partitionnent le problème en sous-problèmes indépendants qu'ils résolvent récursivement, puis combinent leurs solutions pour résoudre le problème initial. Mais la méthode diviser-pour-régner est inefficace si on doit résoudre plusieurs fois le même sous-problème. En programmation dynamique, on se rappelle des sous-problèmes que l'on a résolus et de leurs solutions.

Termes de la suite de Fibonacci

Rappellez vous de la fonction ci-dessous de calcul du n^{ème} terme de la suite de Fibonacci où un appel récursif avec n-1 et n-2 entraînait une complexité en $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

```
def fibonacci (n):
    if n == 0 or n == 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Pour calculer le n^{ème} terme on a besoin du n-1 qui lui-même à besoin du n-2 et n-3, qui seront réutilisés pour le n-2 du n^{ème} terme, etc. Il y a donc beaucoup de calculs dupliqués qui pourraient être évités. La solution est de ne les calculer que s'ils ne l'ont pas encore été.

Donner une première version de l'algorithme où les solutions sont calculées pour les sous-problèmes les plus petits, puis de proche en proche les solutions des problèmes plus gros sont calculées en utilisant le principe d'optimalité des termes de la suite et un tableau pour stocker les solutions.

Donner une deuxième version de l'algorithme où une fonction récursive stocke les résultats intermédiaires dans un tableau. Les nouveaux termes n'étant calculés que s'ils ne l'ont pas encore été.

Rendu de pièces de monnaie

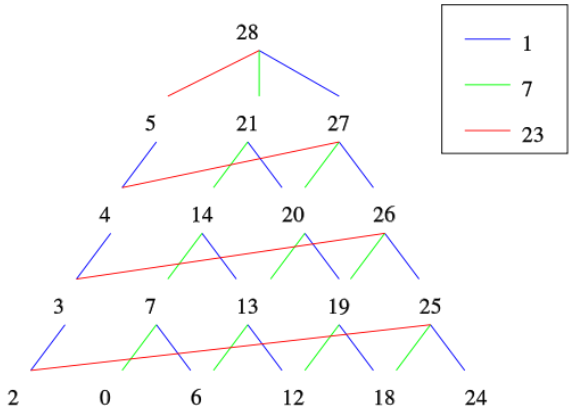
Nous avons vu précédement un algorithme glouton pour résoudre le problème du rendu de pièces de monnaie. La méthode employée était de toujours choisir la pièce de valeur la plus grande possible (afin de se rapprocher de zéro le plus rapidement possible). Nous avons vu que cet algorithme est efficace mais ne rend pas forcément la solution optimale.

```python
systemeMonetaire = [1,2,5,10,20,50,100,200] # exemple de liste de pièces

def renduMonnaie(montant) :
    return renduMonnaieRecur(montant,[], len(systemeMonetaire)-1)

def renduMonnaieRecur(montant,rendu,indPiece) :
    if montant == 0 :
        return rendu
    while montant >= systemeMonetaire[indPiece]:
        montant -= systemeMonetaire[indPiece]
        rendu.append(systemeMonetaire[indPiece])
    return renduMonnaieRecur(montant,rendu,indPiece-1)
```

Un moyen de trouver le nombre de pièces optimal permettant de rendre la monnaie (v) est la programmation dynamique. En effet, supposons qu'on sache rendre de façon optimale toutes les valeurs strictement inférieures à v. Pour rendre v, il faut au moins une pièce, à prendre parmi les n pièces disponibles. Une fois cette pièce choisie, la somme restante est inférieure strictement à v, donc on sait la rendre de façon optimale. Il suffit d'essayer les n possibilités :

$$nbPieces(v) = 1 + \min_{1 \leq i \leq n} nbPieces(v - valeur_i)$$

Prenons par exemple le système monétaire {1, 7, 23} avec lequel nous souhaitons rendre la monnaie sur 28. Un arbre des solutions optimales peut-être construit. Sa racine est la somme à rendre. Un nœud représente une somme intermédiaire. Les fils d'un nœud correspondent aux sommes restantes à rendre selon la dernière pièce rendue (1, 7, ou 23, chaque pièce correspondant à une couleur d'arête attitrée). La construction de l'arbre se fait en largeur d'abord, c'est-à-dire qu'on calcule les fils de tous les nœuds d'un niveau avant de passer au niveau suivant, et on s'arrête dès qu'un nœud 0 est trouvé : le chemin allant de la racine à ce nœud permet de rendre la monnaie avec un minimum de pièces. Ici, on atteint 0 en rendant quatre pièces de 7 (chemin vert de longueur quatre). Le nombre minimal de pièces à rendre est 4 (pièces de 7).



Ecrire une fonction Python qui prend en paramètre le montant à rendre et qui retourne la liste des pièces à rendre suivant le principe de la programmation dynamique. Le système monétaire sera connu et stocké sous la forme d'une liste de valeurs triées par ordre croissant.

Indication : Vous pourrez commencer par écrire une fonction qui crée et retourne un tableau à deux dimensions pieces_min tel que pieces_min[i][j] contient le nombre minimal de pièces pour un rendu de j centimes avec des pièces de valeur inférieure ou égale à celle de la i^{ème} plus petite pièce du système monétaire.

Le sac à dos

Nous avons vu précédement le problème dit du sac à dos. Dans ce problème on dispose d'un ensemble S de n objets. Chaque objet i possède une valeur b_i et un poids w_i. On souhaiterait prendre une partie T de ces objets dans notre sac à dos, malheureusement, ce dernier dispose d'une capacité limitée (en poids) W. On cherche à maximiser la somme des valeurs des objets que l'on peut mettre dans le sac à dos, sans en dépasser la capacité.
Mathématiquement, cela se traduit par :

$$\max_{T \subseteq S} \sum_{i \in T} b_i \quad \text{avec} \quad \sum_{i \in T} w_i \leq W$$

La première idée est d'ajouter les objets de valeurs élevées en premier, jusqu'à saturation du sac.

Prenons l'exemple suivant d'un ensemble S de n=5 objets et d'un sac à dos de capacité W=6.

Objet	A	B	C	D	E
Valeur	6	3	3	3	1
Poids	5	2	2	2	1

Suivons le principe de la méthode et prenons les objets de plus grande valeur d'abord. Ça nous donne le sous-ensemble contenant les objets A et E (valeur totale de 7, poids total de 6). Pourtant, on remarque qu'en choisissant les 3 objets B, C et D on aurait pu atteindre une valeur totale de 9, pour le même poids. L'algorithme n'a pas produit une solution optimale.

Supposons la liste Python `objets` qui contient les objets disponibles sous la forme `[ [ objet1 , valeur1 , poids1 ] , [ objet2 , valeur2 , poids2 ] , ... ]` et triée par ordre décroissant de valeur.

La fonction gloutonne `sacADos` qui prend en argument la capacité W du sac à dos et qui retourne la liste des noms des objets de valeur maximale est la suivante.

```
def sacADos(W) :
   poidsActuel = 0
   solution = []
   for i in range(len(objets)) :
      if poidsActuel + objets[i][2] <= W:
         poidsActuel += objets[i][2]
         solution.append(objets[i][0])
   return solution
```

Cet algorithme glouton n'est pas optimal. Nous allons maintenant nous intéresser à une version optimale reposant sur le principe de la programmation dynamique.
Nous allons stocker la valeur maximale du sac pour les différentes combinaisons d'ensemble d'objets et de capacité restante du sac à dos dans un tableau 2D nommé max_val. En même temps que l'on va remplir ce tableau nous allons remplir un autre tableau de même taille, nommé optimal, qui indique si l'objet fait partie de la configuration optimale courante. Après avoir compléter ces tableaux nous parcourons le tableau optimal afin de trouver tous les éléments du sac à dos faisant partie de la solution optimale globale.

Donner les tableaux max_val et optimal pour l'exemple donné précédement. Indiquer ensuite le chemin que l'on doit parcourir dans le tableau optimal pour trouver la solution finale.

Ecrire la fonction qui permet de créer, et remplir les deux tableaux max_val et optimal, et qui retourne le tableau optimal. Cette fonction prend la capacité du sac à dos en paramètre et affecte chaque case en utilisant le principe d'optimalité du problème.

Ecrire la fonction qui retourne la liste des objets de valeur optimale pour un poids donné en paramètre.