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.

```
def fibonacci(n):
    tableau = [0]*(n+1)
    tableau[1] = 1
    for i in range(2,n+1):
        tableau[i] = tableau[i-1] + tableau[i-2]
    return tableau[n]
```

Ici, les problèmes les plus petits sont les valeurs directes de la suite (0 pour n=0 et 1 pour n=1). On boucle ensuite sur toutes les valeurs de 2 à n, en lisant deux valeurs et affectant une. Chaque valeur est créée qu'une seule fois (et lue deux fois). La complexité est en O(n).

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.

```
def fibonacci(n):
    tableau = [None] * (n+1)
    return fibonacciRec(n,tableau)

def fibonacciRec(n,tableau):
```

```
    if tableau[n]:
        return tableau[n]
    if n == 0 or n == 1:
        tableau[n] = n
    else:
        tableau[n] = fibonacciRec(n-1,tableau) + fibonacciRec(n-2,tableau)
    return tableau[n]
```

Cette version est assez proche de la version récursive de base mais les résultats sont stockés dans un tableau et avant d'effectuer les appels récursifs on vérifie que le terme n'a pas déjà été calculé. Chaque valeur n'est donc calculée qu'une seule fois. La complexité est aussi en O(n).

On peut noter qu'une autre version de cette fonction existe n'utilisant pas le principe de la programmation dynamique et qui est encore meilleure (temps toujours en O(n) mais espace mémoire en O(1)). Ceci est possible car le principe d'optimalité des termes de la suite est très simple (dépendance directe avec les deux termes précédents).

```
def fibonacci (n):
    i = 0
    j = 1
    for k in range(n):
        temp = i + j
        i = j
        j = temp
    return i
```

On peut aussi noter, en analysant la structure des appels récursifs, une autre version récursive qui se base sur le principe que nous n'avons pas besoin de tout mémoriser. Cette version ne fait aussi qu'un seul appel récursif mais cette fois, n'utilise pas un tableau pour stocker toutes les valeurs intermédiaires mais fait juste remonter les deux valeurs précédentes.

```
def fibonacci (n, last = True):
    if n == 0 :
        res = (0,0)
    elif n == 1 :
        res = (0,1)
    else :
        (fib1, fib2) = fibonacci(n-1,False)
        res = (fib2,fib1+fib2)
    if last :
        return res[1]
    else :
        return res
```

Rendu de pièces de monnaie

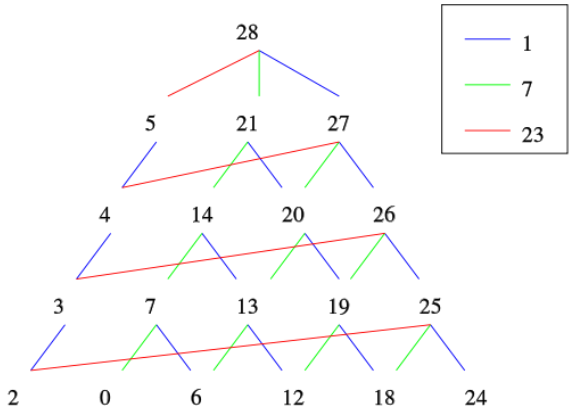
Nous avons vu précédement un algorithme glouton pour résoudre le problème du rendu de pièces de monnaie. La méthode employée était de toujours choisir la pièce de valeur la plus grande possible (afin de se rapprocher de zéro le plus rapidement possible). Nous avons vu que cet algorithme est efficace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

```
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.

```
def creerTableauPiecesMin(montant):
    # n = nombre de pièces différentes
    n = len(systemeMonetaire)
    # init à infini matrice de taille montant (de 0 à montant par pas de 1)
    # par n (nombre de pièces différentes)
    pieces_min = [[float('inf')] * (montant + 1) for _ in range(n)]
    for sousMontant in range(montant + 1): # pour tout sous-montant inférieur à montant
        if sousMontant % systemeMonetaire[0] == 0: # si plus petite pièce diviseur du sous-montant
            pieces_min[0][sousMontant] = sousMontant // systemeMonetaire[0]
            # alors on peut atteindre le sous-montant avec ces petites pieces
    for i in range(1, n):   # on parcours le reste des lignes (pièces)
        for j in range(montant + 1):   # on parcours toutes les colonnes (sous-montants)
            pieces_min[i][j] = pieces_min[i - 1][j]
            # la solution avec des plus petites pièces est aussi une solution avec une plus grosse
            if systemeMonetaire[i] <= j: # si la valeur de la piece est plus petite
                                         # que le sous-montant (ie. on peut utiliser la pièce)
                if pieces_min[i][j - systemeMonetaire[i]] + 1 < pieces_min[i][j]:
                  # si utiliser la pièce i permet de rendre moins de pièces que la solution précédente
                    pieces_min[i][j] = pieces_min[i][j - systemeMonetaire[i]] + 1
                    # alors on l'utilise
    return pieces_min

def renduMonnaieProgDyn(montant):
    pieces_min = creerTableauPiecesMin(montant)
    n = len(systemeMonetaire)
```

```
    # pieces_min[i][m] contient le nombre minimal de pièces rendues pour le montant m
    # avec des pièces de valeur inférieure ou égal à la pièce d'indice i
    # retourner le dernier élément du tableau retournerait le nombre minimal de pièces
    # mais on cherche à aussi récupérer la valeur de chaque pièce rendue
    rendu = []  # la liste des pièces à rendre
    i = n - 1  # i sert à parcourir les lignes du tableau (pièces), de plus grosse à la plus petite
    while montant > 0:   # on aura fini lorsque l'on aura atteint un montant nul à rendre
        if montant >= systemeMonetaire[i]:   # si la pièce i est utilisable
            if pieces_min[i][montant - systemeMonetaire[i]] + 1 == pieces_min[i][montant]:
                # si la pièce i est optimale
                rendu.append(systemeMonetaire[i]) # on ajoute la pièce i à la solution
                montant -= systemeMonetaire[i]  # on décrémente le montant
                continue # on peut passer à la résolution du nouveau montant
        i -= 1  # on ne peut pas utiliser la pièce i (trop grosse ou pas optimale),
                # on passe à la suivante (plus petite)
    return rendu
```

En mémoire, on a besoin d'un tableau 2D de taille (nombre de pièces x montant) ce qui peut-être très grand. La complexité temporelle est identique, l'algorithme nécessite de remplir dans tous les cas, la totalité du tableau 2D donc en $O(nbPieces \times montant)$, même si la deuxième partie de l'algorithme en n'explore qu'une partie. L'optimalité a été obtenue au prix de la complexité.

Remarque. Nous avons vu dans l'énoncé que la recherche de solution pouvait se faire à travers le parcours, en largeur, d'un arbre n-aire (n étant le nombre de pièces disponibles) dont la racine est la somme à rendre. Nous pouvons donc écrire un algorithme dont le but est de parcourir cet arbre et s'arrête dès qu'une somme intermédiaire à rendre vaut 0. Nous savons que pour parcourir en largeur un arbre nous pouvons utiliser une file. La création de l'arbre n'est en fait pas utile nous pouvons simplement « simuler » son parcours en utilisant la file. Cette version permet de ne pas avoir à calculer les solutions pour toutes les configurations (c-à-d pour toutes les cases du tableau précédent) étant donné que beaucoup ne sont jamais considérées.

```
def renduMonnaieProgDynArbre(montant):
    deja_calcule = [False] * montant # pour garder trace des montants déjà calculés
    file = pf.File([(montant, [])])  # la file du parcours en largeur de l'arbre
    while not file.estVide() : # tant que pas tout exploré
        (valeur, liste_pieces) = file.traiter() # on traite le premier de file
        for piece in systemeMonetaire:  # on calcule les n fils de ce noeud
            if valeur == piece: # si on a trouvé la dernière pièce qui donne une valeur à rendre de 0
                liste_pieces.append(piece) # on ajoute la pièce à la solution
                return liste_pieces  # on retourne la solution
            if valeur - piece > 0 and not deja_calcule[valeur - piece]:
            # pas encore fini (nouvelle valeur positive et pas déjà calculée)
                liste_pieces_tmp = liste_pieces.copy()  # copie de la liste
                liste_pieces_tmp.append(piece) # on ajoute la pièce à la solution et on continue
                file.enfiler((valeur - piece, liste_pieces_tmp)) # on ajoute dans la file
                deja_calcule[valeur - piece] = True # on indique qu'on est passé par cette valeur
    print("Rendu impossible") # si on arrive là c'est que le rendu n'est pas possible avec ce sys mon
```

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.

Les deux tableaux sont représentés ici dans un seul où le premier élément est max_val et le deuxième optimal.

	0	1	2	3	4	5	6
0	[0, False]	[0, False]	[0, False]	[0, False]	[0, False]	[0, False]	[0, False]
A	[0, False]	[0, False]	[0, False]	[0, False]	[0, False]	[6, True]	[6, True]
B	[0, False]	[0, False]	[3, True]	[3, True]	[3, True]	[6, False]	[6, False]
C	[0, False]	[0, False]	[3, False]	[3, False]	[6, True]	[6, False]	[6, False]
D	[0, False]	[0, False]	[3, False]	[3, False]	[6, False]	[6, False]	[9, True]
E	[0, False]	[1, True]	[3, False]	[4, True]	[6, False]	[7, True]	[9, False]

Le tableau max_val est initialé à 0 et optimal à False. On remplit les tableaux ligne par ligne (0 à E) et colonne par colonne (0 à 6). La première ligne est forcément à (0, False) : impossible d'avoir un poids sans objet. La première colonne restera forcément à zéro (s'il n'existe pas d'objet de poids nul). Pour chaque case, si le poids de l'objet n'est pas trop grand, on regarde la solution optimale avec un poids identique sans l'objet (case au dessus) sol_{so}, la valeur de l'objet v_o et la solution optimale sans l'objet ni son poids sol_{sonp} (case sur la ligne au dessus et la colonne qui correspond au retrait du poids de

l'objet) et si $sol_{so} < v_o + sol_{sonp}$ alors on a trouvé une nouvelle solution optimale pour ce poids et cet ensemble d'objets (on met à jour les deux tableaux). On fait ça jusqu'à la dernière case (en bas à droite) qui aura comme max_val la valeur maximale globale (i.e. pour la capacité totale du sac à dos et l'ensemble entier des objets).

On commence ensuite le parcours par la case en bas à droite (i.e. on regarde si l'objet E fait partie de la solution), qui est à faux, donc on passe à l'objet précédent (le D). Le D, pour un poids de 6 est à vrai donc il fait partie de la solution finale. On enlève le poids correspondant à D (2) et donc on regarde sur la ligne C et la colonne 6-2=4. L'objet C est à vrai donc on le garde. On passe à la ligne B et le poids 4-2=2. L'objet B est à vrai pour un poids de 2 donc on le garde. On passe à la ligne A et la colonne 2-2=0. L'objet A est à faux pour un poids de 0 (comme tous les objets), on passe à la ligne 0 qui est à faux, et on s'arrête. Les objets sélectionnés sont donc D, C et B, pour une valeur de 9 et un poids de 6. C'est bien une solution optimale à ce problème.

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.

```
def max_valeur(W):
    # init du tableau des valeurs a 0
    max_val = [[0]*(W+1) for _ in range(len(objets)+1)]
    # init du tableau des booleens a False
    optimal  = [[False]*(W+1) for _ in range(len(objets)+1)]
    for i in range(1,len(objets)+1):  # on parcourt/ajoute les objets un à un
        for w in range(W+1):    # on calcule l'optimal pour toutes les capacités possibles
            if objets[i-1][2] > w:
                # si l'objet est plus lourd que la capacité (on ne peut pas le prendre)
                max_val[i][w] = max_val[i-1][w]   # on a la même valeur que sans l'objet
            else:   # si on peut prendre l'objet
                if objets[i-1][1] + max_val[i-1][w - objets[i-1][2]] > max_val[i-1][w]:
                    # si la somme de sa valeur avec la valeur optimale sans l'objet et
                    # son poids est plus grande que la valeur optimale sans l'objet
                    # mais avec son poids, alors on doit garder l'objet
                    max_val[i][w] = objets[i-1][1] + max_val[i-1][w-objets[i-1][2]]
                    optimal[i][w] = True
                else:
                    # sinon on ne garde pas l'objet, la valeur est la même que sans l'objet
                    max_val[i][w] = max_val[i-1][w]
    return optimal
```

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

```
def sacADosProgDyn(W):
    optimal = max_valeur(W)
    # on parcourt le tableau optimal et on construit l'ensemble des objets optimaux
    objets_optimaux = []
    for i in range(len(objets),0,-1):   # on parcourt le tableau du bas vers le haut
        if optimal[i][W]:     # si l'objet est optimal pour la capacité courante
            objets_optimaux.append(objets[i-1][0])   # on l'ajoute à la solution
            W -= objets[i-1][2]   # on diminue la capacité du poids de l'objet
        # en passant à la prochaine itération (i-1) on ne considère plus l'objet i
    return objets_optimaux
```

De la même manière que sur le problème du rendu de monnaie, la complexité est en $O(nombreObjets \times capacité)$. Attention, il ne faut pas conclure à une complexité linéaire par rapport à la capacité du sac. Le nombre de valeurs de W est en effet $\log_2 W$ (si codé en binaire) et on a évidement $W = 2^{\log_2 W}$ donc la complexité est **exponentielle** par rapport à la taille de W. Elle est toutefois toujours meilleure que l'approche naive en 2^n.