PYTHON. **Généralités sur la programmation dynamique**

1) Problématique

Il existe différentes stratégies pour résoudre un problème. Par exemple :

- les algorithmes gloutons (le choix d'une optimisation locale donne finalement la solution optimale)

- les algorithmes de type *diviser pour régner* : dichotomie, QuickSort, ...

Mais ces algorithmes ont des limites :

- Les algorithmes gloutons ne sont généralement pas optimaux

- L'approche *diviser pour régner* est efficace pour des problèmes dont les sous-problèmes sont disjoints. Mais parfois, des sous-problèmes se chevauchent et ont des sous-structures communes. L'approche est alors inefficace puisqu'on résout plusieurs fois les mêmes sous-problèmes.

Afin de résoudre ces problèmes, nous allons utiliser la programmation dynamique.

2) La programmation dynamique

La programmation dynamique est une méthode de construction des solutions par combinaison des solutions des sous-problèmes. Cette approche divise le problème en sous-problèmes de taille moindre, et ne résout chaque sous-problème qu'une seule fois.

Pour pouvoir utiliser la programmation dynamique, un problème doit avoir la propriété de sous-structures optimales : on dit qu'un problème exhibe une sous-structure optimale si, et seulement si une solution optimale au problème contient en elle des solutions optimales à ses sous-problèmes. Par exemple, si dans un graphe on connaît un plus court chemin Γ pour aller d'un sommet x à un sommet y, alors on a un plus court chemin pour tout couple de sommets situés sur Γ.

3) Implémentation par les méthodes de haut en bas et de bas en haut

a) Calcul de haut en bas, avec mémoïsation

On effectue le calcul *de haut en bas* lorsqu'on procède par appels récursifs où on est défini le résultat par appels récursifs sur des sous-structures. C'est souvent la manière la plus simple d'écrire l'algorithme, mais une implémentation naïve peut donner une complexité temporelle catastrophique.

Il est alors primordial de prévoir une structure de données dans laquelle on sauvegarde toutes les solutions de sous-problèmes déjà rencontrés, par une liste ou dans le cas général par un dictionnaire.

Lorsque l'on veut obtenir un résultat pour un sous-problème, on regarde d'abord si il a été déjà calculé, et si c'est le cas on récupère directement la valeur qui a été stockée, sinon, on lance un calcul récursif.

Remarque **La mémoïsation** peut être interprétée comme le stockage de données permettant de ne pas avoir à calculer plusieurs fois la même valeur, afin d'éviter des situations comme par exemple celle obtenue par une définition récursive naïve de la fonction de Fibonacci.

Remarque : Schéma général : On cherche à calculer $f(x)$, qui est une valeur $v(x)$ connue (car x est une structure élémentaire), soit définie comme une fonction F des $f(y)$, où $y \in S(x)$ sont des sous-structures de x.

- sans mémoïsation :

```
def f(x) :
    if (x est élémentaire) :  return v(x)
    return F([f(y) for y in S(x)])
```

- avec mémoïsation :

```
def f(x) :
    dico = {}
    def traite(x) :
        if not(x in dico) :
        if (x est élémentaire) :  dico[x] = v(x)
        else :
            for y in S[x] :  traite(y)
            dico[x] = F([dico[y] for y in S(x)])
    traite(x) ; return dico[x]
```

Attention : Il est essentiel de définir le dictionnaire **en dehors de la procédure récursive** `traite` afin d'opérer sur un **unique dictionnaire** tout au long de l'exécution de l'algorithme.

b) Calcul de bas en haut

On effectue un calcul *de bas en haut* lorsqu'on utilise une programmation itérative, en partant des sous-stuctures élémentaires et en construisant petit à petit les solutions des sous-problèmes de plus en plus grands, jusqu'à arriver au problème qu'on souhaite résoudre.

La mise en œuvre d'un calcul de bas en haut n'est pas toujours aisée.

c) *Remarque culturelle* : Tout algorithme de programmation dynamique peut se ramener à la recherche du plus court chemin dans un graphe.

4) Exemples

a) Le rendu de monnaie

On se donne un jeu de pièces $P \subset \mathbb{N}^*$. Par exemple $P = \{1, 2, 5, 10, 50, 100\}$.

Il s'agit de déterminer pour un entier n le nombre minimum de pièces appartenant à P dont la somme vaut n.

La méthode par programmation dynamique consiste à construire le tableau $(s(k))_{0 \leq k \leq n}$ en notant que

$$s(0) = 0 \text{ et } s(n) = 1 + \min_{p \in P \text{ et } p \leq n} s(n-p)$$

On a $s(0) = 0$, puis on calcule successivement $s(1), ..., s(n)$ en stockant ces valeurs dans le tableau s.

On obtient ainsi une complexité en $O(nm)$, où $m = \operatorname{card} P$.

Remarque : Une programmation récursive sans mémoïsation conduit à une complexité exponentielle.

Programmation de bas en haut :

```
def rendu(n,P) :     #  P est la liste des types de pièces
       M = [0]*(n+1)
       for k in range(1,n+1) :
            v = k     # on a toujours s(k) ≤ k
            for p in P :
                 if p <= k :
                    v = min(v,M[k-p])
            M[k] = v + 1
       return P[n]
```

Programmation de haut en bas :

```
def rendu(n,P) :
       M = [-1]*(n+1)
          ### tableau de mémoïsation et de marquage : M[k] vaut -1 si M[k] n'a pas été calculé
       def traite(k) :     # calcule et définit M[k]
            if M[k] >= 0 :  Return None
            v = k
            for p in P :
                 if p <= k :
                    traite(k-p) ; v = min(v,M[k-p])
            M[k] = v + 1
       traite(n) ; return M[n]
```

b) L'algorithme de Floyd-Warshall pour déterminer les distances entre sommets d'un graphe

La longueur d'un chemin est le nombre d'arêtes qu'il contient. L'algorithme de Floyd calcule *tous* les plus courts-chemins (il y a donc n^2 distances à calculer). Les chemins de longueur minimum sont des chemins réduits (c'est-à-dire ne passant pas deux fois par un même sommet, donc de longueur $\leq n$).

Principe : On numérote les sommets, et on détermine à la k-ième étape la longueur minimale $d_k(x, y)$ des chemins reliant x à y dont *tous les sommets intermédiaires* appartiennent aux k premiers sommets.

D'autre part, on initialise $d_0(x, y) = 1$ s'il existe une arête $x \to y$, et $d_0(x, y) = +\infty$ sinon.

Si z est le k-ième sommet, on a : $\forall (x, y)$, $\boxed{d_k(x, y) = \min(d_{k-1}(x, y), d_{k-1}(x, z) + d_{k-1}(z, y))}$.

On obtient ainsi toutes les longueurs $d(x, y) = d_n(x, y)$ avec une complexité $O(n^3)$.