Chapitre 7

Programmation dynamique

1 Introduction

Définition 1 :

Le programmation dynamique est une stratégie de résolution de problèmes (souvent liés à des problème d'optimisation.)L'idée central est la mémorisation (ou mémoïsation) des solutions des sous-problèmes pour ne pas les recalculer.

Principe de la méthode :
Le but est de résoudre des problèmes d'optimisation.

- On établit une relation de récurrence entre des solutions optimales de sous-problèmes.
- On calcule la valeur de la solution optimale.

Exercice 1 : On considère une suite fini d'entiers relatif : $\{a_1, a_2, ..., a_n\}$.
Le but est de calculer la somme maximale de termes consécutifs de la suite.

Exemple : Pour la liste d'entier, que l'on présente sous forme de tableau :

suite_exemple =	[5, −2, −6, 4, 3, −2, 8, −2, 1, −5]

La liste de somme maximale est : 4 + 3 + (-2) + 8 = 13

1. On procède tout d'abord dans le sens "montant" :
   Pour chaque élément de la suite, on calcule les différentes sommes en partant de cet élément, en conservant la plus forte, puis on conserve la plus forte des plus fortes :
   Pour l'exemple :
   - En partant de 5, la somme maximale est 10.
   - En partant de −2 , la somme maximale est 5.
   - En partant de −6 , la somme maximale est 7.
   - ...

   On propose le script en Python suivant :

```
def somme_max_1(suite):
    """ Somme maximum de termes consecutifs de suite """
    n = len(suite)
    maxi = suite[0]
    for i in range(n):
        for j in range(i, n):
            somme = 0
            for k in range(i, j + 1):
                somme = somme + suite[k]
```

```
            if somme > maxi:
                maxi = somme
    return maxi
```

(a) Justifier que la complexité de `somme_max_1` est $O(n^3)$.

(b) La partie centrale de l'algorithme peut être modifié pour obtenir une complexité moindre. Écrire la fonction `somme_max_2` de complexité $O(n^2)$.

2. **La deuxième approche : La programmation dynamique :**
On note, pour $i \in \{1, ..., n\}$, $M(i)$, la plus grande somme de termes consécutifs de la suite jusqu'à l'indice i inclus :

$$M(i) = \max_{1 \leq k \leq i} \left(\sum_{p=k}^{i} a_p \right)$$

(a) Completer le tableau suivante :

Indice	0	1	2	3	4	5	6	7	8	9
Liste	5	-2	-6	4	3	-2	8	-2	1	-5
M										

(b) Montrer que l'on a la relation :

$$M(1) = a_1 \text{ et } \forall i \in \{1, ..., n-1\}, M(i+1) = \begin{cases} M(i) + a_{i+1} & \text{si} \quad M(i) \geq 0 \\ a_{i+1} & \text{sinon} \end{cases}$$

(c) En déduire la fonction `somme_max_3` de complexité $O(n)$.

Exercice 2 : On dispose d'une matrice M à coefficients entiers (avec n lignes et p colonnes). Le but de l'exercice est de trouver le chemin de poids maximum en partant de $m_{1,1}$, se déplaçant de haut en bas et de gauche à droite, pour arriver à $m_{n,p}$, et en sommant les coefficients rencontrés :

Exemple :

Pour $M = \begin{pmatrix} \underline{2} & 5 & 1 & 4 \\ \underline{6} & \underline{5} & 1 & 3 \\ 3 & \underline{2} & \underline{7} & \underline{1} \end{pmatrix}$ le chemin souligné a pour poids : 23.

1. On note $P = (p_{i,j}) \in \mathscr{M}_{n,p}(\mathbb{R})$ la matrice de poids maximum associé à M, tel que $p_{i,j}$ est le poids maximum des chemins allant de $m_{1,1}$ à $m_{i,j}$.
Montrer que l'on a les relations suivantes, $\forall (i,j) \in [\![1,n]\!] \times [\![1,p]\!]$:
   - $p_{1,1} = m_{1,1}$
   - $p_{1,j} = p_{1,j-1} + m_{1,j}$, pour $j \geq 2$
   - $p_{i,1} = p_{i-1,1} + m_{i,1}$, pour $i \geq 2$
   - $p_{i,j} = \max(p_{i-1,j}, p_{i,j-1}) + m_{i,j}$
2. Écrire la fonction `matrice_poids_max` qui renvoie la matrice P décrite précédemment.
3. En déduire la fonction `poids_max` qui répond au problème.
4. Quelle est la complexité de la fonction `poids_max` ?

Exercice 3 : **Le problème du sac à dos**

Nous disposons de N objets. Chaque objets i possèdent un poids p_i et une valeur v_i.
Nous ne pouvons porter qu'un poids de P.

Le but du problème est de remplir son sac à dos de façon à optimiser la valeurs des objets dans le sac à dos, sous la contrainte du poids maximal.
Le contexte est celui d'un cambrioleur qui ne peut pas porter plus de 10kg.
Il cherche donc à optimiser son butin, sous la contrainte de ce qu'il peut porter.
Lors de son cambriolage, il met la main sur la liste suivante :

Nom de l'objet	poids (p_i)	valeur(v_i)
ordinateurs	5	450
coffre à bijoux	1	250
flute traversière	2	150
sac de montres	1	130
Livre dédicacé	2	170
Console de jeu	4	350
Pile de jeux	2	230
Statuette	4	360
Tableau	2	220
Couvert en argent	3	360

Pour représenter les objets sous python, nous allons utiliser la classe suivante :

```
class mes_objets :
    def __init__(self,nom, poids, valeur ):
        self.nom = nom
        self.poids = poids
        self.valeur = valeur
        self.rapport = valeur/poids

    def __lt__(self,objet):
        """
        relaltion d'ordre <
        """
        return self.rapport < objet.rapport

    def __str__(self):
        return self.nom
```

2 Algorithme glouton

La première approche est avec l'algorithme glouton :

- On trie les objets dans l'ordre décroissant de leur intérêt (ici, c'est le prix au kilo...)
- On prend alors les objets dans cet ordre, tant que l'on peut en prendre...

1. Définir les objets à l'aide de la classe.
2. Écrire la fonction `tri_objet` qui prend en argument la liste des objets, et renvoie la liste ordonnée dans l'ordre décroissant de leur "intérêt".
3. Écrire la fonction `solution_glouton` qui répond au problème du cambrioleur.

3 Algorithme dynamique

On utilise une table de programmation dynamique T deux dimensions telle que $T(i,p)$ représente le profit maximal que l'on peut obtenir en prenant un sous-ensemble d'objets de $\{1,...,i\}$ dont le poids total est inférieur ou égal à p (où p est inférieur à P, le poids maximum et i inférieur ou égal au nombre d'objets, noté N).
Donc $T(i,p)=0$, s'il n'existe aucun sous-ensemble de $\{1,...,i\}$ de poids au plus égal à p.
Évidement, pour tout $0 \leq i \leq N$, on a $T(i,0)=0$.
On dispose aussi d'une relation récursive suivante :

$$\begin{cases} T(i+1,p) &= \max(T(i,p), T(i,p-P_{i+1})+V_{i+1}) & \text{si } P_{i+1} \leq p \\ T(i+1,p) &= T(i,p) & \text{sinon} \end{cases}$$

Autrement dit,$T(i+1,p)$ est obtenu soit sans prendre l'objet $i+1$, il vaut donc $T(i,p)$, soit en ajoutant l'objet $i+1$ au meilleur chargement d'objets pris dans $\{1,...,i\}$ de poids au plus $p-P_{i+1}$

1. Compléter le tableau suivant :

Capacité du sac			0	1	2	3	4	5	6	7	8	9	10
Num	Poids	Valeur	Valeur du sac										
0	5	450	0	0	0	0	0	450	450	450	450	450	450
1	1	250	0	250	250	250	250	450	700	700	700	700	700
2	2	150	0										
3	1	130	0										
4	2	170	0										
5	4	350	0										
6	2	230	0										
7	4	360	0										
8	2	220	0										
9	3	360	0										1230

2. Écrire la fonction `construction_tab` qui prend en argument la liste des objets et le poids maximum, et retourne le tableau défini précédemment.

3. Modifier la fonction précédente pour qu'avec les mêmes arguments, la fonction `solution_dyna` retourne la liste des objets, le poids total et la valeur maximale.

4. Comparer les deux méthodes, en changeant le poids maximal.