Programmation dynamique

La récursivité et la méthode diviser pour régner permet de **résoudre des problèmes de manière descendante** en divisant un exemplaire en sous-exemplaires puis de fusionner ensuite les solutions obtenues sur les sous-exemplaires pour résoudre l'exemplaire de départ.

Par contre pour certains problèmes cette manière de procéder est inefficace car on retombe souvent sur les mêmes sous-exemplaires et on refait toujours les mêmes calculs, ce qui est une perte de temps

La programmation dynamique est une méthode ascendante itérative : On initialise la méthode dans un tableau par les sous-exemplaires les plus petits, puis "par fusion" on obtient les sous-exemplaires plus grands et ainsi de suite on remplit le tableau jusqu'à obtenir la solution de l'exemplaire du départ

1 Exemples

1.1 La suite de Fibonnacci

La suite de Fibonnacci est **définie par récurrence** par
$F_0 = 0, F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geqslant 2$

1.1.1 Une solution récursive

On suit la définition mathématique

```
def fibo(n):
    assert n >= 0
    if n == 0:
       return 0
    if n == 1:
       return 1
    return fibo(n-1)+fibo(n-2)
```

En procédant ainsi, si on exécute..... image

1.1.2 Une solution par programmation dynamique

Dans un tableau T on initialise T[0] avec 0 et T[1] avec 1

```
def fibo_dyn(n:int)->int:
    assert n >= 0
    T = [0]*(n+1)
    T[1] = 1
    for i in range(2,n+1):
        T[i] = T[i-1] + T[i-2]
    return T[n]
```

1.2 Les coefficients binomiaux

Les coefficients binomiaux sont définis par récurrence ainsi :
$\binom{n}{0} = \binom{n}{n} = 1$
$\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$ pour $n \geqslant 2$ et $p \geqslant 1$

1.2.1 Une solution récursive

On suit la définition mathématique

```
def coeff_bin(n,p):
    assert n >= 2 and p >= 0
    if p == 0:
       return 1
    if p == n:
       return 1
    return coeff_bin(n-1,p)+coeff_bin(n-1,p-1)
```

1.2.2 Une solution par programmation dynamique

Une première approche : On utilise un tableau à deux dimensions

```
def coeff_bin_dyn(n:int,p:int)->int:
    assert n >= 1 and p >= 0

    T = [[0]*(n+1) for i in range(n+1)]
    for lig in range(1,n+1):
        T[lig][0],T[lig][lig] = 1,1
        for col in range(1,lig):
            T[lig][col] = T[lig-1][col]+T[lig-1][col-1]
    return T[n][p]
```

Une deuxième approche : On n'utilise que deux listes, on utilise ainsi moins d'emplacement dans la mémoire

```
def coeff_bin_dyn2(n:int,p:int)->int:
    assert n >= 1 and p >= 0

    T1 = [0]*(n+1)
    T1[0],T1[1] = 1,1
    for lig in range(2,n+1):
        T2 = [0]*(n+1)
        T2[0],T2[lig]=1,1
        for col in range(1,lig):
            T2[col] = T1[col]+T1[col-1]
        T1 = T2
    return T1[p]
```

2 Le problème du rendu de monnaie

Imaginons que nous devons rendre 43 euros et que nous avons à notre disposition des pièces de 1,2,5,10 euros **avec la contrainte supplémentaire de rendre le moins de pièces possible (problème d'optimisation)** (pour simplifier il n'y a pas de billets mais que des pièces)

Nous avons vu l'année dernière une stratégie gloutonne pour résoudre ce problème La solution est $43 = 4 \times 10 + 1 \times 2 + 1 \times 1$

Par contre cette stratégie ne marche plus si on change l'ensemble des pièces par exemple :

Imaginons que nous devons toujours rendre 43 euros et que cette fois ci nous avons à notre disposition des pièces de 1,3,7,21,30 euros avec la contrainte supplémentaire de rendre le moins de pièces possible

On rend donc 1 pièce de 30 et une pièce de 7 et deux pièces de 3 euros.

On a donc rendu 4 pièces **mais on peut faire mieux** avec 3 pièces car $43 = 2 \times 21 + 1$

On va chercher une formule de récurrence pour résoudre ce problème de manière générale et de manière ascendante

1. P un ensemble de pièces, $P = \{p_1 <, p_2, ... < p_n\}$ les p_i sont entiers
2. v une valeur entière et strictement positive

Si $v = 0$ alors on prend 0 pièces

Sinon $nb\_pieces(v) = 1 + min(nb\_pieces(v - p))$ où $p \in P$ et $v - p \geqslant 0$

2.0.1 Une solution récursive

```
def nb_pieces(v,P):
    """
    v un entier >= 0
    P une liste d'entiers strictement positifs
    """
    assert v >= 0
    if v == 0:
       return 0
   return 1 + \
   min([nb\_pieces(v-p) for p in P if v-p >= 0])
```

2.0.2 Programmation dynamique

Comment mettre en action la relation de récurrence de manière ascendante ?

```
def nb_pieces_dyn(v,P):
    """
    v un entier >= 0
    P une liste d'entiers strictement positifs
    """
    assert v >= 0
     T = [0]*(v+1)
```

```
    for i in range(1,v+1):
        T[i] = 1 + min([T[i-p] for p in P if v-p >= 0])
    return T[v]
```

3 Le problème du découpage des barres

longueur i	1	2	3	4	5	6	7	8	9	10
prix p_i (euros)	1	5	8	9	10	17	17	20	34	30

Problème Etant donné une barre de longueur n déterminer le revenu maximal r_n que l'on puisse obtenir en coupant la barre puis en vendant les morceaux

Pour $n = 1$ $r_1 = 1$

Pour $n = 2$ soit on coupe la barre en deux morceaux pour un prix de 2 euros soit on ne la coupe pas dans ce cas on a une valeur plus grande 5 euros donc $r2 = 5$

Pour $n = 3$ si on ne coupe pas on a une valeur de 8 euros si on coupe en 3 morceaux on a une valeur de 3 euros soit on coupe en deux morceaux de valeur $r_2 + r_1$ donc de 6 euros

Par conséquent $r_3 = 8$

On a donc une formule de récurrence

$r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1)$ avec $r_1 = 1$

4 Le problème du sac à dos

Problème

Etant donné un sac à dos de contenance maximale $W = 30$ kg il s'agit **d'optimiser en valeur** le contenu du sac à dos en remplissant le sac avec les objets suivants de prix en euros p_i et de poids w_i en kg

Objets	1	2	3	4
p_i (euros)	7	4	3	3
$w_i(kg)$	13	12	8	10

4.1 Approche "gloutonne"

1. Calculer pour chaque objet son prix au kg
2. Ecrire en pseudo-code un algorithme glouton pour résoudre le problème
3. Définir une fonction Python `sac_a_dos_glouton(prix,poids)` où prix et poids sont deux listes de longueur n et chaque objet i tel que $0 \leqslant i \leqslant n-1$ a un prix `prix[i]` et un poids `poids[i]`
4. Sur ce cas particulier quelle est la solution fournie par l'algorithme ?

4.2 Programmation dynamique

Notations :

1. **Données :** prix et poids sont deux listes de longueur n et chaque objet i tel que $0 \leqslant i \leqslant n-1$ a un prix `prix[i]` et un poids `poids[i]`
2. **Résultat :** une liste x_i pour i de 0 à $n-1$ où $x_i = 0$ si on ne met pas l'objet i dans le sac, et $x_i = 1$ si on met l'objet i dans le sac
3. On note $P(i, w)$ le prix maximal obtenu si on charge le sac uniquement avec les objets de 0 à i et pour un poids total inférieur ou égal à w
   Une solution au problème est $P(n-1, W)$ où W est la contenance maximale du sac

On travaille maintenant avec les données suivantes avec $W = 7$

Objets	0	1	2	3
p_i (euros)	100	55	18	70
$w_i(kg)$	5	3	1	4

Remplir le tableau suivant pour P(i,w)

	w = 0	w = 1	w = 2	w = 3	w = 4	w = 5	w = 6	w = 7
i = 0	0	0	0	0	0	100	100	100
i = 1	0	0	0	55	55	100	100	100
i = 2	0							
i = 3	0							

Récurrence

Définition 1. ***Propriété de la sous structure optimale***
On dit qu'un problème vérifie la propriété de la sous-structure optimale si une solution optimale contient en elle des solutions optimales de sous-problèmes
Exemple :
Dans un graphe pondéré le chemin le plus court de la source s à un sommet v passant par w contient le chemin le plus court de s à w et le chemin le plus court de w à v

Preuve
Par l'absurde :
Supposons que $s \xrightarrow{c} v$ soit le plus court et on décompose $s \xrightarrow{c} v$ en
$s \xrightarrow{c_1} w \xrightarrow{c_2} v$
Si $s \xrightarrow{c_1} w$ n'est pas le plus court alors il existe $s \xrightarrow{c_1'} w$ plus court (soit en poids soit en nombre d'arcs) que $s \xrightarrow{c_1} w$ et dans ce cas $s \xrightarrow{c_1'} w \xrightarrow{c_2} v$ est plus court que $s \xrightarrow{c_1} w \xrightarrow{c_2} v$ ce qui est absurde

Théorème 1. *Lorsqu'un problème vérifie la propriété de la sous-structure optimale on peut penser à la programmation dynamique pour le coder par une formule de récurrence*

Explication de la formule de récurrence pour le problème du sac à dos.

Admettons que le problème du sac à dos vérifie la propriété de la sous-structure optimale

Dans ce cas si dans la solution optimale P(i,w) l'objet i s'y trouve alors on peut isoler cet objet de poids `poids[i]` et de prix `prix[i]` et écrire suivant le principe de la sous-structure optimale

$P(i, w) = P(i - 1, w - poids[i]) + prix[i]$

Si l'objet i ne s'y trouve pas c'est parce que :

1. Soit il est trop lourd dans le sens où $w < poids[i]$, dans ce cas $P(i, w) = P(i-1, w)$
2. Soit $P(i - 1, w - poids[i]) + prix[i] < P(i - 1, w)$

D'où la **formule de récurrence** :

$P(i, 0) = 0$ pour tout i

$P(0, w) = 0$ pour w $<$ poids[0] et $P(0, w) = prix[0]$ pour $w \geqslant poids[0]$

Si $w <$ poids$[i]$ alors $P(i, w) = P(i - 1, w)$

Sinon $P(i, w) = \max(P(i - 1, w), P(i - 1, w - p[i]) + \text{prix[i]}))$

Programmer une fonction Python `sac_a_dos(prix,poids)` qui retourne le prix maximal

5 Le problème de l'édition

6 Exercices

Ex 1

Utiliser le module `time` et chronométrer le temps d'exécution de

1. `fibo(20)` et `fibo(30)`
2. `fibo_dyn(20)` et `fibo_dyn(30)`
3. `coeff_bin(10,2)` et `coeff_bin(30,6)`
4. `coeff_bin_dyn(30,6)` et `coeff_bin_dyn(100,20)`
5. `coeff_bin_dyn2(30,6)` et `coeff_bin_dyn2(100,20)`

Ex 2

Utiliser le module `time` et chronométrer le temps d'exécution de nb_pieces(v) et nb_pieces_dyn(v)

Ex 3

1. Définir une fonction récursive `nb_barres(n)` qui retourne le nombre optimal de barres
2. Définir une fonction itérative `nb_barres_dyn(n)` qui retourne le nombre optimal de barres obtenu par programmation dynamique