LYCÉE LECONTE DE LISLE

Programmation dynamique

Vincent Picard

1
Comprendre la programmation dynamique

Qu'appelle-t-on programmation dynamique ?

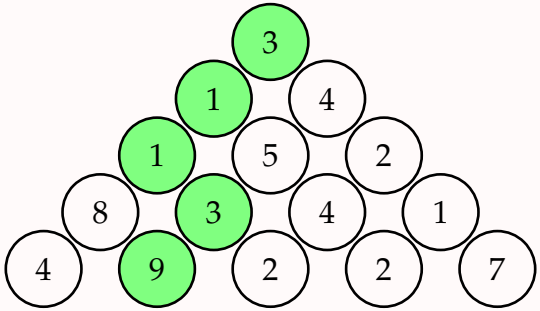
Un problème de programmation dynamique possède les caractéristiques suivantes :

- C'est le plus souvent un **problème d'optimisation**
- Le problème possède la propriété de **sous-structure optimale**
- Il y a **chevauchement** des sous-problèmes

Ces caractéristiques conduisent à des méthodes de résolution spécifiques appelées **programmation dynamique**.

Illustrons ces propriétés sur un problème simple.

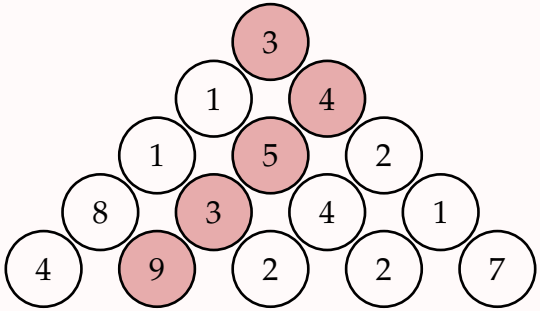
Chemin de somme maximale dans une pyramide



un chemin possible de somme 17

- Un chemin part du sommet de la pyramide et à chaque étape on peut descendre soit à gauche, soit à droite.
- On cherche le chemin de somme maximale : **problème d'optimisation**

La solution optimale

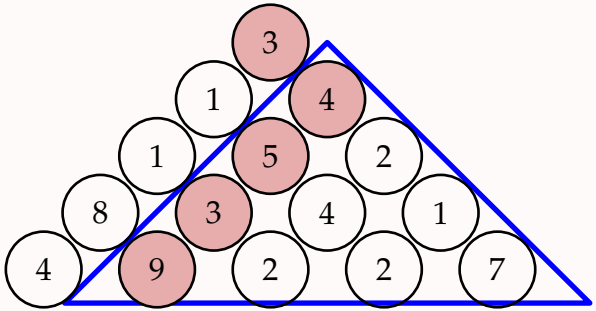


le seul chemin optimal de somme 24

- Considérer les 2^n chemins possibles est de complexité exponentielle : envisageable seulement pour les très petites valeurs de n.
- Un **algorithme glouton** est efficace mais ne permet pas d'aboutir à la solution optimale.

Sous-structure optimale

Un problème possède une **sous-structure optimale** lorsque des parties de la solution optimale sont elles-même des solutions optimales à des sous-problèmes du problème initial.



- Ici le **sous-chemin** 4+5+3+9 est une solution optimale au **sous-problème** obtenu avec la **sous-pyramide** de droite.
- On remarque qu'il en est de même pour 5+3+9, 3+9, ...

Équation de Bellman

- La sous-structure optimale permet d'obtenir une équation de récurrence appelée **équation de Bellman** (chaque problème possède *sa propre* équation de Bellman).
- Ré-organisons les données sous forme matricielle :

$$A = \begin{pmatrix} 3 & & & & \\ 1 & 4 & & & \\ 1 & 5 & 2 & & \\ 8 & 3 & 4 & 1 & \\ 4 & 9 & 2 & 2 & 7 \end{pmatrix}$$

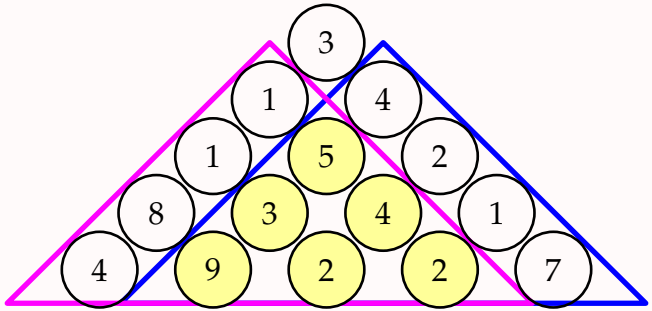
- Notons $S_{i,j}$ la somme maximale au sous-problème correspondant à la sous-pyramide de sommet $a_{i,j}$, alors :

$$S_{i,j} = a_{i,j} + \max\left(\underbrace{S_{i+1,j}}_{\text{sous-pyramide gauche}} , \underbrace{S_{i+1,j+1}}_{\text{sous-pyramide droite}} \right)$$

- **cas d'arrêt :** $S_{i,j} = a_{i,j}$ sur la dernière ligne

Chevauchement des sous-problèmes

$$S_{i,j} = a_{i,j} + \max(S_{i+1,j}, S_{i+1,j+1})$$



En jaune les sous-problèmes communs aux sous-pyramides gauche et droite

Il y a **chevauchement des sous-problèmes** : les sous-problèmes utiles pour répondre au problème initial partagent eux-même des sous-problèmes en commun.

- L'utilisation **directe** de l'équation de récurrence est **inefficace** ! On va recalculer de multiples fois les solutions aux mêmes sous-problèmes.

Solution 1 : récursivité + mémoïsation

On utilise un **dictionnaire** pour mémoriser les réponses aux sous-problèmes déjà résolus.

```
# retourne la somme maximale
def pyramide(A: [[int]]) -> int:
    n = len(A) # nbr de lignes
    dico = {} # pour la memoisation

    def S(i, j): #fonction récursive
        if (i, j) in dico: # si déjà calculé
            return dico[(i, j)]
        else : # sinon on le calcule
            if i == n-1: # cas d'arrêt
                x = A[i][j]
            else:
                x = A[i][j] + max(S(i+1,j), S(i+1, j+1))
            dico[(i, j)] = x # sauvegarde du resultat
            return x

    return S(0, 0)
```

Solution 2 : construction bottom-up

On résout **tous** les sous-problèmes en commençant par les petits et en remontant vers le problème initial, en **tabulant** chaque résultat.

```
# retourne la somme maximale
def pyramide(A: [[int]]) -> int:
    n = len(A) # nbr de lignes
    # Une matrice pour tabuler les résultats :
    S = [[0 for j in range(n)] for i in range(n)]

    for j in range(n):
        S[n-1][j] = A[n-1][j] # cas de base

    for i in range(n-2, -1, -1): # lignes de bas en haut
        for j in range(i+1):
            S[i][j] = A[i][j] + max(S[i+1][j], S[i+1][j+1])

    return S[0][0] # solution au problème initial
```

Reconstruction du chemin

- Les deux solutions proposées fournissent grâce à la programmation dynamique la valeur de la somme maximale.
- Si l'on veut de plus savoir le chemin utilisé il faut **reconstruire la solution optimale**.
- Cette reconstruction ne peut se faire qu'en sachant quels sous-problèmes ont été utiles pour aboutir à la solution : ici, savoir si le max a été obtenu à gauche ou à droite.
  - **Avec la solution 1 :** on peut faire en sorte que la fonction récursive retourne aussi la solution optimale
  - **Avec la solution 2 :** il faut tabuler en plus les sous-chemins optimaux OU plus efficace, tabuler si la solution vient de gauche ou de droite et utiliser un autre algorithme pour reconstruire la solution...

Reconstruction du chemin par mémoïsation

```
# retourne la somme maximale et un chemin optimal
def pyramide(A: [[int]]):
    n = len(A) # nbr de lignes
    dico = {} # pour la memoisation

    def S(i, j): #fonction récursive
        if (i, j) in dico: # si déjà calculé
            return dico[(i, j)]
        else : # sinon on le calcule
            if i == n-1: # cas d'arrêt
                x = A[i][j]
                chemin = [(i, j)]
            else:
                sg, cg = S(i+1, j)
                sd, cd = S(i+1, j+1)
                if sg > sd :
                    x = A[i][j] + sg
                    chemin = [(i, j)] + cg
                else:
                    x = A[i][j] + sd
                    chemin = [(i, j)] + cd
            dico[(i, j)] = (x, chemin) # sauvegarde du resultat
            return (x, chemin)

    return S(0, 0)
```

2
Le problème du sac à dos

Un problème d'optimisation

- On possède un sac à dos dont la capacité est un entier $C \in \mathbb{N}$.
- On considère un ensemble $E = \{x_1, x_2, \dots x_n\}$ de n objets qu'il est possible de placer dans le sac.
- Chaque objet x possède :
  - un **poids** entier positif, noté $p(x)$
  - une **valeur**, notée $v(x)$ positive (non nécessairement entière)
- On cherche à choisir le meilleur sous-ensemble d'objets X à placer dans son sac, de telle sorte à ce que le poids total n'excède pas la capacité du sac à dos et que la valeur totale soit la meilleure possible.
- Mathématiquement, on veut donc trouver $X \subset E$ tel que :

$$\sum_{x \in X} p(x) \leq C$$

$$\sum_{x \in X} v(x) \text{ est maximal}$$

Une instance du problème

- Voici une instance du problème du sac à dos :
  - Capacité : $C = 5$
  - Ensemble d'objets :

Objet	Poids	Valeur
x_1	5	24
x_2	3	15
x_3	3	15
x_4	2	12
x_5	2	12

- Sur ce petit exemple, on trouve assez rapidement "à l'œil" le sac à dos $X = \{x_2, x_4\}$ de poids $5 \leq C$ et de valeur 27 qui semble optimale.
- On remarque qu'il n'y a pas nécessairement unicité de la solution : $X = \{x_3, x_5\}$ convient également.

La résolution exhaustive

- La première idée est d'essayer tous les sacs à dos possibles, c'est-à-dire considérer tous les sous-ensembles X de E, puis de garder le meilleur sac à dos convenable trouvé.
- Le problème est le nombre de sac à dos possible : si on décide de choisir k objets alors il y a $\binom{n}{k}$ manières de choisir ces k objets. Comme on peut choisir entre 0 et n objets, il y aurait donc au total

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

  sac à dos à envisager.
- Cette solution donnerait un algorithme de complexité temporelle exponentielle $O(2^n)$: seuls les instances avec un petit nombre d'objets sont traitables en temps raisonnable.
- Par exemple, si l'ordinateur met $1s$ pour considérer un sac à dos et qu'il y a 30 objets à choisir. Il faudrait environ 32 ans pour résoudre le problème, pour 50 objets c'est environ 30 millions d'années qui seront nécessaires...

Utiliser un algorithme glouton

- Vous avez vu en première année des algorithmes reposant sur une stratégie gloutonne : on construit la solution par étapes, à chaque étape on fait un choix localement optimal, on espère que la solution construite sera optimale.
- Ici, on peut envisager 3 manières d'appliquer une stratégie gloutonne :
  - Choisir les objets de meilleur valeur d'abord : dans l'exemple on obtient $X = \{x_1\}$ de valeur 24 qui n'est pas optimale...
  - Choisir les objets de plus petit poids d'abord : dans l'exemple on obtient $X = \{x_4, x_5\}$ de valeur 24 qui n'est pas optimale...
  - Choisir les objets de meilleur rapport valeur/poids d'abord : dans l'exemple on obtient $X = \{x_4, x_5\}$ de valeur 24 qui n'est pas optimale...
- En bref, la stratégie gloutonne peut fournir rapidement une solution de bonne valeur mais elle ne garantit pas que la solution est optimale !

Calcul des rapports valeur / poids

x	$p(x)$	$v(x)$	$v(x)/p(x)$
x_1	5	24	4,8
x_2	3	15	5
x_3	3	15	5
x_4	2	12	6
x_5	2	12	6

Sous-structure optimale

- Si on considère un **sac à dos optimal** qui contient un objet y, alors nécessairement $X \setminus \{y\}$ est une solution optimale au problème du sac à dos où la capacité vaut $C - p(y)$ et où l'ensemble d'objets prenable est $E \setminus \{y\}$.
- Autrement dit, les solutions au problème du sac à dos contiennent en elle-même des solutions optimales à des sous-problèmes du problème initial.
- On dit alors que le problème exhibe une **sous-structure optimale** : c'est un bon indicateur pour déterminer que la programmation dynamique est adaptée à la résolution du problème.

Équation de Bellman

- Lorsqu'on décide de résoudre le problème par programmation dynamique il faut commencer par établir une relation de récurrence appelée **équation de Bellman**.
- Pour $k \in \{0, \dots, n\}$ et $m \in \{0, \dots, C\}$, notons $f(k, m)$ la valeur d'un sac à dos optimal n'utilisant que les k premiers objets à disposition et une capacité de sac à dos m.
- On remarque qu'on a deux possibilités pour construire un tel sac optimal
  - Soit la solution contient le k-ème objet et le reste du sac est une solution optimale à un sous-problème, donc :

$$f(k, m) = v(x_k) + f(k-1, m - p(x_k))$$

  - Soit la solution ne contient pas le k-ème objet et alors c'est aussi une solution au sous-problème où on ne considère que les $k-1$ premiers objets :

$$f(k, m) = f(k-1, m)$$

- Dans tous les cas, on a donc l'équation de Bellman suivante :

$$f(k,m) = \begin{cases} \max\{f(k-1,m) \quad , \quad v(x_k) + f(k-1, m-p(x_k))\} & \text{si } p(x_k) \leq m \\ f(k-1,m) & \text{sinon} \end{cases}$$

- On souhaite calculer $f(n,C)$: on peut prendre tous les objets avec une capacité de C.
- Comme pour le problème des escaliers de Fibonacci on va procéder des petits sous-problèmes aux grands et tabuler les résultats dans une matrice pour éviter de les recalculer.

Mise en œuvre de la programmation dynamique

- **Cas de base de la récursivité**
  - $f(0, \bullet) = 0$: si on ne peut prendre aucun objet la valeur sera nulle.
  - $f(\bullet, 0) = 0$: si la capacité est 0 la valeur sera nulle.
- **Ordre dans lequel les calculs seront faits** : il y a plusieurs ordres possibles de calcul des $f(k, m)$.
  - Ici, on voit que pour calculer les $f(k, \bullet)$ on a besoin des valeurs de $f(k-1, \bullet)$, donc on décide de traiter les problèmes par k croissants puis par capacité croissante.
  - Dans la matrice qui va suivre, cela correspond à un remplissage ligne par ligne.

Résolution par programmation dynamique bottom-up

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	24
2	0	0	0	15	15	24
3	0	0	0	15	15	24
4	0	0	12	15	15	27
5	0	0	12	15	24	27

La dernière case en bas à droite correspond à la valeur de $f(n, C)$ ce qui vaut bien 27 comme on l'avait deviné !

Comment le programmer ?

```
poids = [0, 5, 3, 3, 2, 2]
valeur = [0, 24, 15, 15, 12, 12]

n, C = 5, 5
T = [[0 for j in range(C+1)] for i in range(n+1)]

for k in range(1, n+1):
    for m in range(1, C+1):
        if poids[k] <= m:
            sans = T[k-1][m]
            avec = valeur[k] + T[k-1][m - poids[k]]
            if avec > sans:
                T[k][m] = avec
            else:
                T[k][m] = sans
        else:
            T[k][m] = T[k-1][m]

print("La valeur optimale est ", T[n][C])
```

Complexité

- On calcule chaque case de la matrice une et une seule fois donc la complexité de cette programmation dynamique est en $O(nC)$.
- Cela signifie que si la capacité du sac est fixée une fois pour toute, le temps de calcul croit linéairement en fonction du nombre d'objets (et non plus exponentiellement comme c'était le cas avec la méthode exhaustive).

Oui mais quels objets dois-je prendre ?

- Pour l'instant la programmation dynamique utilisée permet uniquement de connaître la valeur d'un sac à dos optimal, mais elle ne nous dit pas quels objets prendre...
- Pour faire cela, il faut se rappeler pour chaque calcul du choix fait pour le max : en effet on se rappelle que l'un des choix signifiait de prendre l'objet et l'autre de ne pas le prendre.

Mise en évidence du chemin utilisé pour calculer la valeur optimale

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	24
2	0	0	0	15	15	24
3	0	0	0	15	15	24
4	0	0	12	15	15	27
5	0	0	12	15	24	27

On voit que les moments où on a fait le choix "avec" correspondent aux 4e et 2e objets. Donc le sac à dos optimal construit est $X = \{x_2, x_4\}$.

Programme en mémorisant les informations de reconstruction

```
poids = [0, 5, 3, 3, 2, 2]
valeur = [0, 24, 15, 15, 12, 12]

n = 5
C = 5

# Calcul avec informations pour la reconstruction
T = [[0 for j in range(C+1)] for j in range(n+1)]
U = [[False for j in range(C+1)] for j in range(n+1)]

for k in range(1, n+1):
    for m in range(1, C+1):
        if poids[k] <= m:
            sans = T[k-1][m]
            avec = valeur[k] + T[k-1][m - poids[k]]
            if avec > sans:
                T[k][m] = avec
```

```
                U[k][m] = True
            else:
                T[k][m] = sans
                U[k][m] = False
        else:
            T[k][m] = T[k-1][m]
            U[k][m] = False

print(T)
print(U)
```

Programme de reconstruction de la solution

Si on a mémorisé les informations de reconstruction, on peut reconstruire la solution optimale depuis la matrice : en partant de la dernière case et en remontant jusqu'au cas de base :

```
i, j = n, C # coordonnées de la case actuelle

res = []
while (i > 0 and j > 0): # tant que ce n'est pas un cas de base
    if U[i][j]: # si on a fait le choix de prendre l'objet
        res.append(i) # ajouter i dans la solution
        j = j - poids[i] # remonter
        i = i - 1
    else:
        i = i - 1 # remonter

print(res)
```

Exercice : le sac à dos par mémoïsation

1. Écrire une fonction `sacados(poids, valeur, capacite)` qui retourne la valeur optimale d'un sac à dos en utilisant la programmation dynamique avec memoisation.

2. Adapter la fonction précédente pour qu'elle retourne la liste des couples (poids, valeur) à choisir.

3
Plus longue sous séquence commune

Sous-séquences

- Soit $u = u_1 u_2 \dots u_n$ une séquence de n éléments (par exemple une suite de lettres). On appelle **sous-séquence** de u une suite extraite v de u. Formellement, on dit que $v = v_1 v_2 \dots v_m$ est une sous-séquence de u s'il existe une application $\varphi : \{1, \dots, m\} \to \{1, \dots, n\}$ strictement croissante telle que $v = u_{\varphi(1)} u_{\varphi(2)} \dots u_{\varphi(m)}$.

- Par exemple, pour la séquence `MARMAILLE` on peut trouver les sous séquences : `MAMIE`, `MARIE`, `ARME`, `MALLE`, *etc*.

- Attention, il est interdit de changer l'ordre des éléments de la séquence initiale pour produire une sous-séquence.

Le problème de la plus longue sous séquence commune

- Soit deux séquences u et v deux séquences de longueurs respectives n et m.
- On cherche à déterminer une séquence w vérifiant :
  - w est une sous-séquence de u
  - w est une sous-séquence de v
  - w est de longueur maximale

Résolution exhaustive

- Pour résoudre le problème, on pourrait :

  1. Produire la liste L_1 des sous-séquences de u
  2. Produire la liste L_2 des sous-séquences de v
  3. Calculer la liste L des éléments de $L_1 \cap L_2$
  4. Déterminer un plus long élément de L

- L'inconvénient est qu'il existe 2^n sous-séquences de u (pour chaque lettre on peut décider de la prendre ou pas, cela fait 2 possibilités pour chaque lettre).

- Cette solution a donc une complexité exponentielle et n'est pas utilisable en pratique sauf pour les petits cas.

Sous-structure optimale

- Si U et V sont deux séquences de longueur n et m respectivement, on notera $U[1..i]$ le préfixe de longueur i de U et $V[1..j]$ le préfixe de longeur j de V.
- Ainsi pour chaque couple (i, j) avec $0 \leq i \leq n$ et $0 \leq j \leq m$ on obtient un sous-problème du problème initial.
- Notons $P(i, j)$ la longueur de la PLSC à $U[1..i]$ et $V[1..j]$. On cherche à déterminer $P(n, m)$.
- Vérifions que le problème possède une sous-structure optimale

Équation de Bellman

- On remarque tout d'abord que si $U[i] = V[j]$ alors une PLSC de $U[1..i]$ et $V[1..j]$ s'obtient en prenant la dernière lettre de chacun des mots et en déterminant une plus longue sous-séquence commune à $U[1..i-1]$ et $V[1..j-1]$. Autrement dit :

$$\forall i > 0, \forall j > 0,\ P(i,j) = 1 + P(i-1, j-1)$$

- Si $U[i] \neq V[j]$ alors une PLSC de $U[1..i]$ et $V[1..j]$ est soit une PLSC de $U[1..i-1]$ et $V[1..j]$ ou alors une PLSC de $U[1..i]$ et $V[1..j-1]$ car la plus longue sous séquence ne peut pas choisir à la fois $U[i]$ et $V[j]$ comme dernière lettre. Autrement dit :

$$\forall i > 0, \forall j > 0,\ P(i,j) = \max\{P(i-1, j), P(i, j-1)\}$$

- Il reste à déterminer les cas de base :
  - $\forall j, P(0,j) = 0$ (on ne peut prendre aucune lettre dans U)
  - $\forall i, P(i,0) = 0$ (on ne peut prendre aucune lettre dans V)

Résolution

- Résolvons la programmation dynamique pour U = IMPOSSIBLE et V = MEPRISAIT.

	.	M	E	P	R	I	S	A	I	T
.	0	0	0	0	0	0	0	0	0	0
I	0									
M	0									
P	0									
O	0									
S	0									
S	0									
I	0									
B	0									
L	0									
E	0									

Résolution (solution)

- Résolvons la programmation dynamique pour U = IMPOSSIBLE et V = MEPRISAIT.

	.	M	E	P	R	I	S	A	I	T
.	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	1	1	1	1	1
M	0	1	1	1	1	1	1	1	1	1
P	0	1	1	2	2	2	2	2	2	2
O	0	1	1	2	2	2	2	2	2	2
S	0	1	1	2	2	2	3	3	3	3
S	0	1	1	2	2	2	3	3	3	3
I	0	1	1	2	2	3	3	3	4	4
B	0	1	1	2	2	3	3	3	4	4
L	0	1	1	2	2	3	3	3	4	4
E	0	1	2	2	2	3	3	3	4	4

Reconstruction de la solution

- Comme pour le sac à dos, on part de la valeur finale calculée et on remonte pour comprendre les choix qui ont été effectués :

	.	M	E	P	R	I	S	A	I	T
.	0	0	0	0	0	0	0	0	0	0
I	0	0	0	0	0	1	1	1	1	1
M	0	1	1	1	1	1	1	1	1	1
P	0	1	1	2	2	2	2	2	2	2
O	0	1	1	2	2	2	2	2	2	2
S	0	1	1	2	2	2	3	3	3	3
S	0	1	1	2	2	2	3	3	3	3
I	0	1	1	2	2	3	3	3	4	4
B	0	1	1	2	2	3	3	3	4	4
L	0	1	1	2	2	3	3	3	4	4
E	0	1	2	2	2	3	3	3	4	4

- On remarque les cas qui correspondent à prendre une lettre correspondent aux lettres MPSI : c'est donc la PLSC cherchée.

Exercice : programmation bottom-up

1. Écrire une fonction `PLSC(u: str, v: str) -> int` retournant la longueur de la PLSC à u et v.

2. Modifier la fonction pour qu'elle retourne un couple (M, ℓ) où ℓ est la longueur de la PLSC et M est une matrice telle que $M[i, j]$ contienne une valeur (1, 2 ou 3) renseignant sur le cas de calcul rencontré pour la case (i, j).

3. Ecrire une fonction `reconstruit(u, v, M)` qui reconstruit la PLSC de u et v à partir des informations de reconstruction M.

Résolution de PLSC par mémoïsation

```
def PLSC(u : str, v: str):
    dico = {} # Pour mémoriser les cas résolus

    def P(i, j):
        if (i, j) in dico:
            return dico[(i, j)]
        else:
            if i == 0 or j == 0:
                res = 0
            else:
                if u[i-1] == v[j-1]:
                    res = 1 + P(i-1, j-1)
                else:
                    res = max(P(i-1,j), P(i, j-1))
            dico[(i, j)] = res # On mémorise
            return res
    return P(len(u), len(v))
```

Résolution de PLSC par mémoïsation : avec reconstruction

```
def PLSC(u : str, v: str):
    dico = {} # On mémorise maintenant des couples (longueur, ss-seq commune)

    def P(i, j):
        if (i, j) in dico:
            return dico[(i, j)]
        else:
            if i == 0 or j == 0:
                res = 0
                seq = ''
            else:
                if u[i-1] == v[j-1]:
                    (l, s) = P(i-1, j-1)
                    res = 1 + l
                    seq = s + u[i-1]
                else:
                    (l1, s1) = P(i-1, j)
                    (l2, s2) = P(i, j-1)
                    if l1 >= l2:
                        res = l1
```

```
                    seq = s1
                else:
                    res = l2
                    seq = s2
        dico[(i, j)] = (res, seq) # On mémorise
        return (res, seq)
return P(len(u), len(v))
```

Bottom-up ou mémoïsation ?

- Laquelle des deux solutions de programmation dynamique est préférable ? Cela dépend du problème étudié.
- Si l'équation de Bellman conduit à calculer tous (ou presque) les sous-problèmes, c'est-à-dire **le chevauchement est important**: procéder de bas en haut pour éviter l'utilisation inutile du dictionnaire **[Exemple : PLSC]**
- Sinon, si le calcul ne nécessite pas de résoudre tous les sous-problèmes, c'est-à-dire **le chevauchement est faible**, utiliser la mémoïsation peut économiser les calculs en évitant de résoudre les sous-problèmes qui n'interviennent pas **[Exemple : Sac à dos]**