COURS : PROGRAMMATION DYNAMIQUE
= PLUS LONGUE SOUS-SUITE COMMUNE =

Notre quatrième étude de cas concerne le problème du calcul de la plus longue sous-suite commune (en anglais : Longest Common Subsequence ou LCS). Ce problème classique de programmation dynamique permet de mesurer la similarité entre deux séquences. Nous allons construire la solution du problème par programmation dynamique.

I) DÉFINITION DU PROBLÈME

Une instance du problème est spécifiée par deux chaînes de caractères : une chaîne S1 de longueur n et une chaîne S2 de longueur m.

Une sous-suite (on dit aussi sous-séquence) d'une chaîne est obtenue en supprimant zéro, un ou plusieurs caractères, sans changer l'ordre des caractères restants (contrairement à une sous-chaîne, elle n'est pas forcément contiguë).

Par exemple, si S = "informatique", alors "info", "format", "ique" sont des sous-suites (pas forcément uniques), et "iatq" aussi (on garde l'ordre, mais on saute des lettres).

La plus longue sous-suite commune de S1 et S2 est une sous-suite qui est sous-suite de S1 et de S2 et qui a une longueur maximale.

On note généralement LCS(S1, S2) une plus longue sous-suite commune (la séquence), et ℓ(S1, S2) sa longueur.

Problème de la plus longue sous-suite commune (LCS)

Entrée : Deux chaînes de caractères S1 et S2.

Sortie : Un entier ℓ (S_1, S_2), égal à la longueur de la plus longue sous-suite commune de S1 et S2.

Exemple : Soient S1 = "ABC" et S2 = "BAC".
Les sous-suites de "ABC" incluent : "", "A", "B", "C", "AB", "AC", "BC", "ABC". Celles de "BAC" incluent : "", "B", "A", "C", "BA", "BC", "AC", "BAC". Les sous-suites communes les plus longues ont longueur 2 : par exemple "AC" ou "BC". Donc ℓ(S1, S2) = 2.

La plus longue sous-suite commune a de nombreuses applications pratiques :

- Bio-informatique : comparer des séquences ADN ou protéiques pour détecter des similarités évolutives ;
- Comparaison de fichiers : l'algorithme diff (utilisé par Git) s'appuie sur la LCS pour identifier les lignes modifiées ;
- Détection de plagiat : mesurer la similarité structurelle entre deux textes ;
- Correction orthographique : suggérer des corrections en trouvant des parties communes.

Pour résoudre ce problème de manière exhaustive (par brute force), il faudrait tout d'abord lister toutes les sous-suites possibles de S1, puis pour chacune, vérifier si elle est aussi une sous-suite de S2, et enfin garder la plus longue.

Une chaîne de longueur n possède 2^n sous-suites (chaque caractère peut être inclus ou non). Vérifier si une sous-suite de longueur k est présente dans S2 se fait en O(m). Dans le pire des cas, le nombre de vérifications à effectuer est donc de $O(m \cdot 2^n)$. C'est un problème exponentiel qui demande une approche plus efficace.

II) SOUS-STRUCTURE OPTIMALE ET RELATION DE RÉCURRENCE

Pour appliquer la programmation dynamique, on doit identifier des sous-problèmes pertinents et une relation de récurrence entre eux, issue de la structure d'une solution optimale.

II.1. Sous-structure optimale

Considérons une instance du problème avec une chaîne S1[1..n] et une chaîne S2[1..m]. Regardons les derniers caractères des deux préfixes pour établir la sous-structure optimale.

Deux cas sont possibles pour traiter les derniers caractères des préfixes S1[1..n] et S2[1..m] :

Cas n°1 : Les derniers caractères sont identiques (match) : S1[n] == S2[m]
Si S1[n] == S2[m], alors ce caractère commun fait nécessairement partie d'une LCS optimale. En effet, si on avait une LCS qui n'utilisait pas ce caractère commun, on pourrait l'ajouter à la fin et obtenir une sous-suite commune plus longue.

Rechercher la LCS de S1[1..n] et S2[1..m] revient donc à rechercher la LCS de S1[1..n-1] et S2[1..m-1], puis à ajouter ce caractère commun :

$$LCS\ (S_1[1..n], S_2[1..m]) = LCS\ (S_1[1..n-1], S_2[1..m-1]) + 1$$

Exemple : S1 = "ABC" et S2 = "ADC"
- Les derniers caractères sont identiques ('C' == 'C').
  ⇒ LCS("ABC", "ADC") = LCS("AB", "AD") + 1.

Cas n°2 : Les derniers caractères sont différents (pas de match) : S1[n] ≠ S2[m]
Si S1[n] ≠ S2[m], alors au moins l'un des deux derniers caractères ne fait pas partie de la LCS. On a donc deux sous-cas à explorer :

Sous-cas 2a : Le dernier caractère de S1 ne fait pas partie de la LCS. On cherche alors la LCS de S1[1..n-1] et S2[1..m].

Sous-cas 2b : Le dernier caractère de S2 ne fait pas partie de la LCS. On cherche alors la LCS de S1[1..n] et S2[1..m-1].

La LCS optimale est le maximum de ces deux possibilités :

$$LCS\ (S_1[1..n], S_2[1..m]) = max\begin{cases} LCS(S_1[1..n-1], S_2[1..m]) \\ LCS(S_1[1..n], S_2[1..m-1]) \end{cases}$$

Exemple : S1 = "ABC" et S2 = "ABD"
- Les derniers caractères sont différents ('C' ≠ 'D').
  ⇒ LCS("ABC", "ABD") = max{LCS("AB", "ABD"), LCS("ABC", "AB")}.

II.2. Équation de récurrence sur les valeurs optimales

On note $L_{i,j}$ la longueur de la LCS entre les i premiers caractères de S1 et les j premiers caractères de S2.

Remarque : Si S1[i] == S2[j], alors toute sous-suite commune optimale pour S1[1..i] et S2[1..j] peut être supposée se terminer par ce caractère commun. On obtient donc directement

$$L_{i,j} = L_{i-1,j-1} + 1$$

Il est alors inutile de calculer les deux autres candidats $L_{i-1,j}$ et $L_{i,j-1}$: ils correspondent à « ignorer » l'un des deux caractères, ce qui ne peut pas donner mieux qu'utiliser ce match et prolonger une solution optimale sur les préfixes (i-1, j-1).

Récurrence sur la valeur de la solution optimale

Pour tout i ∈ {0..n} et j ∈ {0..m}, les cas de base sont :
- $L_{0,j} = 0$ (la chaîne vide n'a aucun caractère commun avec S2),
- $L_{i,0} = 0$ (S1 n'a aucun caractère commun avec la chaîne vide).

Pour tout i ∈ {1..n} et j ∈ {1..m} :

$$L_{i,j} = \begin{cases} L_{i-1,j-1} + 1 & si\ S1[i] == S2[j] \quad (cas\ n°1\ -\ match) \\ max \begin{cases} L_{i-1,j} \\ L_{i,j-1} \end{cases} & sinon \qquad\qquad (cas\ n°\ 2\ -\ pas\ de\ match) \end{cases}$$

III) SOUS-PROBLÈMES ET COMPLEXITÉ

III.1. Définition des sous-problèmes

Ici, les sous-problèmes sont indexés par les paramètres i (longueur du préfixe de S1, de 0 à n) et j (longueur du préfixe de S2, de 0 à m). Quand i = 0 ou j = 0, S1[1..0] et S2[1..0] sont des chaînes vides.

En faisant varier ces deux paramètres sur toutes les valeurs pertinentes, nous obtenons nos sous-problèmes :

Sous-problèmes de la plus longue sous-suite commune

Calculer $L_{i,j}$, la longueur de la LCS entre le préfixe de longueur i de S1 et le préfixe de longueur j de S2

(Pour chaque i = 0, 1, 2 … n et j = 0, 1, 2 … m)

Le plus grand sous-problème (avec i=n et j=m) est exactement le problème original.

III.2. Schéma de récursion

Le schéma de récursion complet sur un exemple où on cherche la LCS entre S1 = "on" et S2 = "bon" est donné sur la figure ci-dessous. Comme on le verra après, l'algorithme top-down ne calculera pas tous ces cas mais ils sont montrés ici pour illustrer le problème dans son ensemble :

- La notation ["on","bon"] signifie qu'on cherche la longueur optimale pour S1 = "on" et S2 = "bon";
- La notation L[2][3] = 2 signifie que la longueur de la LCS entre les 2 premiers caractères de S1 et les 3 premiers de S2 vaut 2 ;
- Dans le cas n°1 (branche de gauche), à partir des cas de base (en vert), on remonte la valeur L[i-1][j-1] + 1 si S1[i] == S2[j] ;
- Dans les deux sous-cas n°2a et n°2b (branches de droite), à partir des cas de base (en vert), on remonte la valeur L[i-1][j] (sous-cas 2a) et L[i][j-1] (sous-cas 2b).
- Les valeur L[i][j] prennent le maximum des valeurs remontées.

https://www.informatique-f1.fr/dp/SousSuiteCommune/

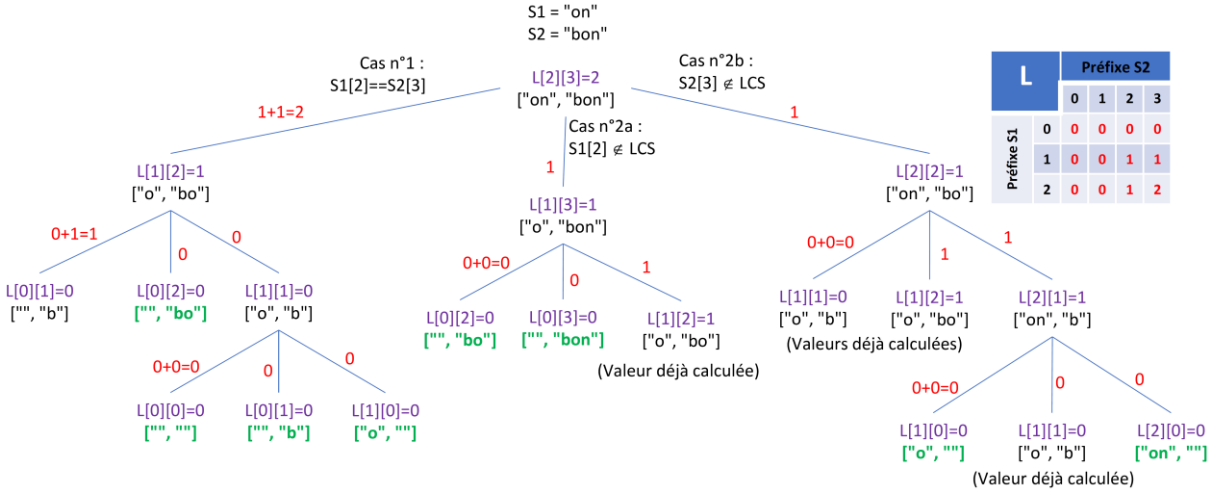

Figure 1 : Schéma de récursion du problème de la sous-suite maximale

III.3. Complexité sans mémoïsation

À chaque étape, on diminue i ou j (ou les deux). La profondeur de récursion est donc au plus (n + m).

Dans le pire des cas (quand S1[i] ≠ S2[j] presque partout), chaque nœud se ramifie en 2, ce qui donne un arbre de récursion pouvant contenir jusqu'à $O(2^{n+m})$ nœuds (ordre de grandeur exponentiel).

Le travail local à chaque nœud est en O(1) (comparaisons, max, +1), donc l'algorithme récursif sans mémoïsation est exponentiel.

Remarquons cependant qu'il n'y a que (n+1)·(m+1) sous-problèmes distincts. De nombreux appels récursifs portent donc sur les mêmes sous-problèmes, d'où l'intérêt de la mémoïsation.

IV) ALGORITHMES DE PROGRAMMATION DYNAMIQUE

IV.1. Algorithme top-down

On mémorise les valeurs déjà calculées $L_{i,j}$ dans un dictionnaire, afin de ne jamais recalculer deux fois le même sous-problème.

Algorithme top-down pour le calcul des valeurs optimales

```
Entrée : S1[1, …, n] : Chaîne S1
S2[1, …, m] : Chaîne S2

Sortie : ℓ (S1, S2)

# Dictionnaire de mémoïsation
L := {}

rec_opt_val_LCS (i, j) :
    # i : longueur du préfixe de S1
    # j : longueur du préfixe de S2

    # Utilise la mémoïsation
    Si (i, j) est dans L :
        Retourner L[(i, j)]

    # Cas de base i ==0 ou j == 0
    Si i == 0 ou j == 0:
        L[(i, j)] := 0
        Retourner L[(i, j)]

    # Cas 1 (match)
    Si S1[i] == S2[j] :
        L[(i, j)] := rec_opt_val_LCS(i - 1, j - 1) + 1
    Sinon
        # Cas 2a et 2b
        V1 := rec_opt_val_LCS (i - 1, j)
        V2 := rec_opt_val_LCS (i, j - 1)
        L[(i, j)] := max (V1, V2)

    Retourner L[(i, j)]

# Appel initial
résultat := rec_opt_val_LCS (n, m)
```

Avec cet algorithme, seuls les cas réellement utiles sont calculés (voir la remarque en page 4 sur l'équation de la récurrence)

Le schéma de récursion en page suivante montre quels sont les cas réellement calculés dans le même exemple que précédemment ainsi que les valeurs enregistrées dans la table de mémoïsation.

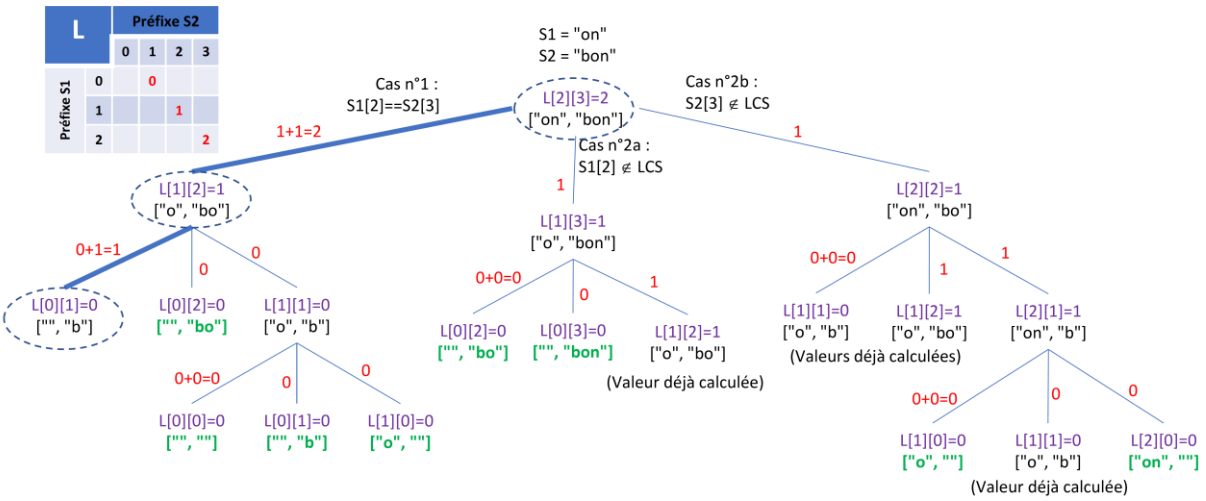

Figure 2 : Application de l'algorithme top-down et table de mémoïsation associée

IV.2. Complexité de l'algorithme top-down

Les états possibles sont les couples (i, j) avec $0 \le i \le n$ et $0 \le j \le m$, soit au plus $(n+1)\cdot(m+1)$, c'est-à-dire $O(n \cdot m)$ sous-problèmes.

Avec la mémoïsation, chaque (i, j) est calculé au plus une fois, et le calcul effectue un travail local en O(1). La complexité en temps est donc $O(n \cdot m)$.

L'espace nécessaire pour sauvegarder les informations dans le dictionnaire est $O(n \cdot m)$.
La profondeur de récursion est au plus (n + m), donc la pile est en O(n + m). L'espace total est dominé par le dictionnaire : $O(n \cdot m)$.

IV.3. Algorithme bottom-up

L'algorithme bottom-up consiste à remplir progressivement la table des solutions des sous-problèmes en utilisant la relation de récurrence, en partant des cas de base.

Algorithme bottom-up pour le calcul des valeurs optimales

```
opt_val_LCS (i, j) :
    # Cas de base i ==0 ou j == 0
    Pour j allant de 0 à m :
    |   L[(0, j)] := 0
    Pour i allant de 0 à n :
    |   L[(i, 0)] := 0

    # Remplissage de la table
    Pour i allant de 1 à n :
    |   Pour j allant de 1 à m :
    |   |   Si S1[i] == S2[j] :
    |   |   |   L[(i, j)] := L[(i - 1, j - 1)] + 1
    |   |   Sinon :
    |   |   |   L[(i, j)] := max(L[(i - 1, j)], L[(i, j - 1)])
    Retourner L[(n, m)]
```

IV.4. Complexité de l'algorithme bottom-up

L'algorithme bottom-up calcule toutes les cases (i, j) pour $0 \le i \le n$ et $0 \le j \le m$, soit exactement $(n+1)\cdot(m+1)$ calculs, chacun en O(1). La complexité en temps et en espace est donc $O(n \cdot m)$.

V) ALGORITHME DE RECONSTRUCTION

V.1. Principe et algorithme de reconstruction

Jusqu'ici, on a calculé la longueur $\ell(S1, S2)$. L'objectif est maintenant de reconstruire une plus longue sous-suite commune (pas seulement sa longueur).

On peut reconstruire une LCS en retraçant un chemin depuis L[n][m] jusqu'à L[0][0] :

- Si S1[i] == S2[j], alors ce caractère appartient à une LCS : on l'ajoute, puis on va en (i−1, j−1).
- Sinon, on regarde quel voisin (i−1, j) ou (i, j−1) conserve la valeur optimale, et on se déplace vers lui.

Remarque : la LCS n'est pas forcément unique. En cas d'égalité entre L[i−1][j] et L[i][j−1], différents choix de déplacement peuvent conduire à différentes LCS, toutes de longueur maximale.

Algorithme de reconstruction

Entrée : S1[1, …, n] : Chaîne S1
S2[1, …, m] : Chaîne S2
L = {(i, j) : … }: Dictionnaire / table des valeurs optimales
Sortie : Seq[…] : Une LCS (liste de caractères)

```
Reconstruction (S1, S2, D) :
    Seq := []
    i := n
    j := m

    Tant que i > 0 et j > 0 :
        # Cas du match (prioritaire)
        Si S1[i] == S2[j] et L[(i, j)] == L[(i - 1, j - 1)]  + 1:
            Ajouter S1[i] à la liste Seq
            i := i - 1
            j := j - 1
        # Cas 2a
        Sinon si L[(i - 1, j)] >= L[(i, j - 1)]
            i := i - 1
        # Cas 2b
        Sinon :
            j := j - 1

    Retourner Seq renversée
```

Remarque importante : avec l'algorithme top-down proposé, certaines valeurs voisines peuvent ne pas exister dans le dictionnaire. En effet, en cas de match, seul L[i-1][j-1] a été calculé, tandis que L[i-1][j] et L[i][j-1] n'existent pas.

Pour garantir la compatibilité, l'algorithme de reconstruction doit tester le cas du match en priorité.

La figure ci-dessous illustre ce principe de reconstruction avec S1 = "on" et S2 = "bon" :

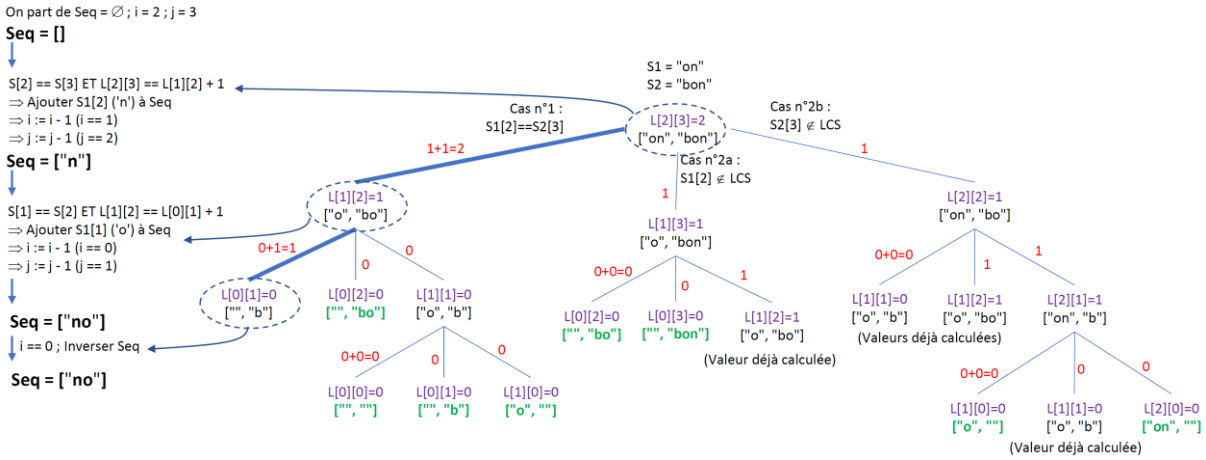

Figure 3 : Principe de reconstruction de la solution optimale

V.2. Complexité finale

La reconstruction parcourt au plus (n + m) étapes (à chaque étape, on diminue i ou j ou les deux), avec un travail local en O(1). Le temps de reconstruction est donc de O(n + m) et l'espace de reconstruction est O(n + m) si on stocke la séquence reconstruite.

La complexité totale (calcul + reconstruction) est donc de O(n·m) + O(n + m) = O(n·m).

Si on ne veut que la longueur ℓ(S1, S2) (pas la reconstruction), on peut réduire l'espace de O(n·m) à O(min(n, m)) en ne gardant que la ligne (ou colonne) précédente. En revanche, pour reconstruire une sous-suite, il faut en général conserver davantage d'informations (table complète ou informations de parenté).