

Devoir surveillé n°1

Durée : 45 minutes

L'utilisation d'une calculatrice ou de tout document est interdite.

Toute sortie anticipée est interdite.

Le sujet s'intéresse au problème de détermination de la longueur de la plus longue sous-séquence commune à deux séquences d'ADN. Cette longueur est un indicateur de proximité d'espèces permettant de les comparer lors d'une étude phylogénétique.

Les parties 1, 2 et 3 sont indépendantes. On pourra utiliser les fonctions de la partie 1 dans la partie 3.

1. Questions préliminaires

- a. Écrire une fonction `maximum` qui renvoie le plus grand des deux nombres passés en argument.

Par exemple, `maximum(2, 4)` devra renvoyer 4.

- b. Écrire une fonction `zeros` qui prend en argument deux entiers naturels n et p (supposés non nuls) et qui renvoie une liste de n listes contenant chacune p zéros, représentant ainsi la matrice nulle à n lignes et p colonnes.

Par exemple, `zeros(2, 3)` devra renvoyer `[[0,0,0], [0,0,0]]`.

2. Plus longue sous-chaîne commune

On considère $A = a_1 \dots a_n$ et $B = b_1 \dots b_p$ deux chaînes de caractères non vides.

On appelle **sous-chaîne** de A toute chaîne de caractères $a_{i_1} \dots a_{i_k}$ où $1 \leq i_1 < \dots < i_k \leq n$ (**ces caractères ne sont pas nécessairement consécutifs** dans A).

On appelle **plus longue sous-chaîne commune** à A et B toute sous-chaîne commune à A et B de longueur maximale. Si l'une des chaînes A ou B est vide, ou si A et B n'ont aucune sous-chaîne commune, on convient que la chaîne vide est l'unique plus longue sous-chaîne commune à A et B .

On s'intéresse alors au problème ci-dessous.

Étant donné deux chaînes de caractères A et B de longueurs respectives n et p , quelle est la longueur d'une plus longue sous-chaîne commune à A et B ?

Par exemple, les chaînes de caractères "AAA" et "TAA" sont des plus longues sous-chaînes communes aux chaînes de caractères `chaine1 = "ATAGA"` et `chaine2 = "TAACA"`.

La chaîne de caractères `sschaine = "ATGC"` est une plus longue sous-chaîne commune aux chaînes de caractères `chaine1 = "AATGCG"` et `chaine2 = "TATTAGC"`.

- a. Quelle est la longueur d'une plus longue sous-chaîne commune aux chaînes "AATGCG" et "TATTAGC" ? Justifier.
- b. Déterminer une plus longue sous-chaîne commune aux chaînes "AATGCG" et "TATTAGC" autre que "ATGC".

- c. Parmi les chaînes de caractères ci-dessous, indiquez sur votre copie quelle est la seule plus longue sous-chaîne commune aux chaînes "TCGTA" et "CTG" ? *On ne demande pas de justifier la réponse.*

"CTG"

"TGCT"

"CGT"

"CG"

- d. Déterminer toutes les plus longues sous-chaînes communes aux chaînes "TCGTA" et "CTG". *On ne demande pas de justifier la réponse.*
- e. Parmi les fonctions ci-dessous, déterminer les deux seules permettant de déterminer si une chaîne de caractères **ssch** est une sous-chaîne d'une chaîne de caractères **ch**. *On ne demande pas de justifier la réponse.*

```
def estSousChaine1(ch,ssch):
    n = len(ch)
    p = len(ssch)
    i, j = 0, 0
    while i < n and j < p:
        if ch[i] == ssch[j]:
            j += 1
        i += 1
    return j == p
```

```
def estSousChaine2(ch,ssch):
    n = len(ch)
    p = len(ssch)
    i, j = 0, 0
    while i < n and j < p:
        if ch[i] == ssch[j]:
            j += 1
        i += 1
    return i == j
```

```
def estSousChaine3(ch,ssch):
    n, p = len(ch), len(ssch)
    for i in range(n):
        j = 0
        while j < p and ch[i+j]==ssch[j]:
            j += 1
        if j == p:
            return True
    return False
```

```
def estSousChaine4(ch,ssch):
    j = 0
    for i in range(len(ch)):
        if ch[i] == ssch[j]:
            j += 1
        if j == len(ssch):
            return True
    return False
```

- f. Écrire alors une fonction booléenne **sousChaineCommune** qui prend en argument trois chaînes de caractères **chaine1**, **chaine2** et **sschaine** qui renvoie **True** si **sschaine** est une sous-chaîne commune à **chaine1** et **chaine2**, et **False** dans le cas contraire.

3. Recherche d'une solution par programmation dynamique

Si $A = a_1 \dots a_n$ et $B = b_1 \dots b_p$ sont deux chaînes de caractères non vides, on note $\ell_{i,j}$ la longueur d'une plus longue sous-chaîne commune aux chaînes $a_1 a_2 \dots a_i$ et $b_1 b_2 \dots b_j$ (respectivement composées des i premiers et j premiers caractères de A et B).

On peut montrer que la suite double $(\ell_{i,j})_{(i,j) \in \llbracket 0,n \rrbracket \times \llbracket 0,p \rrbracket}$ vérifie l'initialisation et la relation de récurrence :

$$\forall (i,j) \in \llbracket 0,n \rrbracket \times \llbracket 0,p \rrbracket, \ell_{i,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ 1 + \ell_{i-1,j-1} & \text{sinon si } a_i = b_j \\ \max(\ell_{i-1,j}, \ell_{i,j-1}) & \text{sinon si } a_i \neq b_j \end{cases}.$$

On cherche donc à calculer la longueur d'une plus longue sous-chaîne commune à $A = a_1 \dots a_n$ et $B = b_1 \dots b_p$, c'est-à-dire le coefficient $\ell_{n,p}$.

Le principe est de calculer de proche en proche chaque coefficient $\ell_{i,j}$ (en respectant l'ordre défini par la relation de récurrence) en les mémorisant dans une matrice.

- a. Quelle est la taille (nombres de lignes et de colonnes) de la matrice où on mémorisera les coefficients $\ell_{i,j}$?
- b. Recopier et compléter le code de la fonction `lplsch` calculant la longueur d'une plus longue sous-chaine commune à deux chaînes de caractères `chaine1` et `chaine2` passées en arguments.

```
def lplsch(chaine1, chaine2):
    n, p = len(chaine1), len(chaine2)
    l = zeros(....., .....)
    for i in range(....., n+1):
        for j in range(1, .....):
            if chaine1[i-1] == chaine2[j-1]:
                l[i][j] = .....
            else:
                l[i][j] = maximum(....., .....)
    return l[.....][.....]
```

- c. Donner la matrice des coefficients ($\ell_{i,j}$) dans le cas où `A="CTG"` et `B="TCGT"`.
- d. Expliquer (sans l'implémenter) comment adapter l'algorithme donné par la fonction `lplsch` pour construire une plus longue sous-chaine commune.
- e. On sait qu'une chaîne de n caractères admet au plus 2^n sous-chaines distinctes (en comptant la chaîne vide). Expliquer pourquoi la méthode programmée à la question précédente est plus efficace qu'en comparant chaque sous-chaine de A avec chaque sous-chaine de B .

Corrigé

1. Questions préliminaires

1.

```
def maximum(a,b):  
    if a>=b:  
        return a  
    else:  
        return b
```

2.

```
def zeros(n,p):  
    L=[]  
    for i in range(n):  
        L.append([0]*p)  
    return L
```

Remarque : une autre syntaxe possible est la suivante, mais celle-ci engendre des problèmes d'aliasing qui s'avéreraient extrêmement néfastes pour la suite.

```
def zeros(n,p):  
    return [[0]*p]*n
```

On peut en revanche écrire les choses de la façon suivante, sans problèmes d'aliasing :

```
def zeros(n,p):  
    return [[0]*p for i in range(n)]
```

2. Plus longue sous-chaîne commune

1. L'énoncé dit qu'une plus longue chaîne commune est "ATGC" donc la longueur d'une plus longue sous-chaîne commune est 4.
2. On a vu dans la question précédente que "AAGC" est une sous-chaîne commune de longueur maximale.
3. La plus longue sous-chaîne commune est "CG".
4. Les plus longues sous-chaînes communes à "TCGTA" et "CTG" sont "CT", "CG" et "TG".
5. Les fonctions qui conviennent sont `estSousChaine1` et `estSousChaine4`.
6. En nommant `estSousChaine` l'une des deux fonctions précédentes, on peut définir la fonction `sousChaineCommune` de la manière suivante :

```
def sousChaineCommune(chaine1, chaine2, sschaine):  
    return estSousChaine(chaine1, sschaine) and  
           estSousChaine(chaine2, sschaine)
```

3. Recherche d'une solution par programmation dynamique

1. La matrice est de taille $(n + 1) \times (p + 1)$ car i varie entre 0 et n et j varie entre 0 et p .
- 2.

```
def lplsch(chaine1, chaine2):
    n, p = len(chaine1), len(chaine2)
    l = zeros(n+1, p+1)
    for i in range(1, n+1):
        for j in range(1, p+1):
            if chaine1[i-1] == chaine2[j-1]:
                l[i][j] = 1+l[i-1][j-1]
            else:
                l[i][j] = maximum(l[i-1][j], l[i][j-1])
    return l[n][p]
```

Remarque : étant donné qu'à la fin de l'exécution i est égal à n et j est égal à p , on peut remplacer la dernière ligne par `return l[i][j]` mais ce n'est pas le plus naturel.

3. La matrice est
$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 2 \\ 0 & 1 & 1 & 2 & 2 \end{pmatrix}.$$

4. La fonction `lplsch` se réfère aux longueurs des chaînes. Une idée possible est de reprendre l'algorithme en remplaçant les longueurs des chaînes par les chaînes elle-mêmes.

Ainsi, on remplace la fonction `maximum` par une fonction `plch` qui prend en argument deux chaînes `chaine1` et `chaine2` et qui renvoie la plus longue (avec le choix arbitraire de renvoyer `chaine1` si les deux chaînes ont la même longueur).

```
def lplc(chaine1, chaine2):
    if len(chaine1) >= len(chaine2):
        return chaine1
    else:
        return chaine2
```

Ensuite, on remplace la fonction `zeros` par une fonction `chaines_vides` qui prend en argument deux entiers naturels non nuls n et p et qui renvoie la liste de n listes contenant chacune p chaînes vides.

```
def chaines_vides(n,p):
    L=[]
    for i in range(n):
        L.append([""]*p)
    return L
```

Enfin, on remplace dans la fonction `lplsch` les longueurs par les chaînes elles-mêmes :

```
def lplsch2(chaine1, chaine2):
    n, p = len(chaine1), len(chaine2)
    l = chaines_vides(n+1, p+1)
    for i in range(1, n+1):
        for j in range(1, p+1):
            if chaine1[i-1] == chaine2[j-1]:
                l[i][j] = l[i-1][j-1] + chaine1[i-1]
            else:
                l[i][j] = lplc(l[i-1][j], l[i][j-1])
    return l[n][p]
```

On crée ainsi une matrice `l` (en fait une liste de listes) dont le coefficient d'indices i et j n'est plus la longueur maximale $\ell_{i,j}$ d'une sous-chaîne commune à $a_1 \dots a_i$ et $b_1 \dots b_j$ mais une sous-chaîne de longueur maximale commune à $a_1 \dots a_i$ et $b_1 \dots b_j$. Ainsi, à la fin de l'exécution `l[n][p]` contient sous-chaîne de longueur maximale commune à $A = a_1 \dots a_n$ et $B = b_1 \dots b_p$.

Par exemple, l'instruction

```
print(lplsch2("TCGTA", "CTG"))
```

donne à l'affichage

```
[['', '', '', ''], ['', '', 'T', 'T'], ['', 'C', 'T',
    'T'], ['', 'C', 'T', 'TG'], ['', 'C', 'CT', 'TG'],
    ['', 'C', 'CT', 'TG']]
```

5. Considérons deux chaînes A et B de longueurs n et p avec, par exemple, $n \geq p$. La méthode par programmation dynamique demande $(n+1)(p+1) \leq (n+1)^2$ tests. Tester si chaque sous-chaîne de A est une sous-chaîne de B demande 2^n tests d'égalité de sous-chaînes (chaque test d'égalité pour une sous-chaîne de longueur k demandant lui-même au moins k tests, un pour chaque caractère de la sous-chaîne).

Dans le premier cas, on a donc un nombre de tests polynomial en n alors que dans le second cas, on a un nombre de tests exponentiel en n . Pour n suffisamment, la première méthode est donc bien plus efficace que la seconde (par croissance comparée, $\frac{(n+1)^2}{2^n} \xrightarrow{n \rightarrow +\infty} 0$).

On pourra remarquer cependant si $n \geq p$, on peut seulement tester si les sous-chaînes de B sont des sous-chaînes de A , ce qui réduit le nombre de tests d'égalité de chaînes à 2^p mais qui nécessite ensuite, pour comparer les deux méthodes, une analyse plus fine concernant n et p , ce qui n'était certainement pas attendu dans cette épreuve !