CORRIGÉ CCINP INFO TSI 2020

1 Génération d'une séquence d'ADN

Q1 -

```
seq = 'ATCGTACGTACG'
print(seq[3])
print(seq[2:6])
```

```
G
CGTA
```

Q2 -

```
import random

def generation(n):
    seq = ''
    codage = [ '', 'A', 'C', 'G', 'T' ]
    # on pouvait aussi utiliser un dictionnaire: codage = { 1:'A', 2:'C', 3:'G', 4:'T' }
    for _ in range(n):
        seq += codage[random.randint(1,4)]
    return seq

print(generation(50))
```

```
GCGTAGACGAAATCGTATCTTGTAGCGAAGAGCTGTGCTAACAGGTGAAA
```

Q3 - La fonction **mystere** commence par compter le nombre d'occurrences des lettres 'A', 'C', 'G' et 'T' dans la chaîne **seq**, puis renvoie une liste formée des pourcentages d'apparition de ces lettres.

Rem : je ne commente pas la façon dont est écrite cette fonction...

Q4 - La boucle **while** est faite autant de fois qu'il y a de caractères dans la chaîne, donc si n est la longueur de **seq**, la complexité est en $O(n)$.

La variable **i** est une variable entière qui diminue de 1 à chaque passage, donc la boucle se termine.

2 Recherche d'un motif

2.1 Algorithme naïf

Q5 - Il est difficile de savoir si l'énoncé permet ou non d'utiliser le *slicing*, c'est-à-dire la possibilité d'extraire directement une sous-chaîne d'une autre à l'aide d'une instruction de la forme **seq[debut:debut+ℓ]**. Je pense que oui, compte tenu de la première question où le slicing apparaît. Néanmoins, je propose deux versions de la fonction **recherche**, car seule la seconde version permettra d'évaluer la complexité demandée à la question suivante.

```
def recherche1(M, T):
    # si on utilise le slicing Python
    m = len(M)
    for i in range(len(T) - m + 1):
        if T[i:i+m] == M:
            return i
    return -1

def recherche2(M, T):
    # si on n'utilise pas le slicing Python
    m = len(M)
    for i in range(len(T) - m + 1):
        j = 0
        while j < m and T[i+j] == M[j]:
```

```
                j += 1
            if j == m:
                return i
        return -1

print(recherche2('BLA', 'blablaBLA'))
print(recherche2('Blab', 'blablaBLA'))
print(recherche2('la', 'blablaBLA'))
```

```
6
-1
1
```

Q6 - Le calcul qui suit s'appuie sur la fonction **recherche2** ci-dessus.

Si le motif de longueur m n'apparaît jamais dans la chaîne de longueur n, il y aura eu $n-m+1$ boucles de faites sur la variable **i**, chacune étant formée de m comparaisons des caractères de M.

À chaque passage dans la boucle **while** intérieure, il y a 2 tests et l'opération **j += 1**, soit 3 opérations, donc un total de $3m$ opérations au maximum pour cette boucle (si le motif apparaît, sinon il y en aura moins). Et à chaque passage dans la boucle **for** extérieure, il y a 1 affectations et une comparaison en plus, soit $3m+2$ opérations par passage, et donc en tout $(3m+2)(n-m+1)$ opérations, dans le pire des cas (je pense que le jour du concours, on pouvait se contenter d'une estimation moyenne de nm opérations, puisque seul compte l'ordre de grandeur).

Pour $m = 50$ et $n = 3 \cdot 10^9$, cela fait 455 999 992 552 opérations.

Si l'ordinateur réalise 10^{12} opérations par seconde, il lui faudra donc environ $0,46$ secondes.

Q7 - Il y a $n-m+1$ morceaux de taille m dans une séquence de taille n. Si l'on recherche chacun de ces morceaux dans une autre séquence d'ADN par la fonction **recherche** précédente, cela fera donc $(3m+2)(n-m+1)^2$ opérations.

Avec les données de l'énoncé, cela fait environ $1,37 \cdot 10^{21}$ opérations, qui seront donc effectuées en environ 1 367 999 955 secondes soit un peu plus de 43 années (il y a 31 536 000 secondes dans une année non bissextile).

La conclusion s'impose d'elle même !

2.2 Algorithme de Knuth-Morris-Pratt

2.2.1 Préfixe et suffixe

Q8 - Les préfixes de 'ACGTAC' sont :

'A', 'AC', 'ACG', 'ACGT' et 'ACGTA'.

Les suffixes de 'ACGTAC' sont :

'C', 'AC', 'TAC', 'GTAC' et 'CGTAC'.

Q9 - Le plus grand préfixe de 'ACGTAC' qui est aussi un suffixe est : 'AC'.

Le plus grand préfixe de 'ACAACA' qui est aussi un suffixe est : 'ACA'.

2.3 Algorithme de Knuth-Morris-Pratt

Q10 - La fonction **fonctionannexe** renvoie une liste (plus précisément une liste d'entiers compris entre 0 et m, où m est la longueur de la chaîne passée en paramètre).

Q11 - Il y a trois erreurs dans la procédure proposée, la première ce ces erreurs n'étant pas une erreur de syntaxe mais une **NameError** :

- ligne 5 : utilisation de la variable m, qui n'a pas été définie. Il faut faire précéder la boucle de l'instruction : **m = len(M)**.
- ligne 6 : **=** au lieu de **==** pour le test d'égalité.
- ligne 10 : manque **:** après **else**.

Q12 - Ci-dessous, j'ai modifié **fonctionannexe** afin qu'elle affiche les valeurs de i, j et F à la fin de chaque boucle, ce qui fournit la réponse à la question posée.

```
def fonctionannexe(M):
    F = [0]
    i = 1
    j = 0
    m = len(M) # erreur variable m utilisée sans être définie
    while i < m:
        if M[i] == M[j]: # erreur = au lieu de ==
            F.append(j+1)
            i = i+1
            j = j+1
        else:  # erreur manque :
            if j > 0:
                j = F[j-1]
            else:
                F.append(0)
                i = i+1
        print('i=', i, 'j=',j, 'F=',F)
    return F

F = fonctionannexe('ACA²ACA')
```

```
i= 2 j= 0 F= [0, 0]
i= 3 j= 1 F= [0, 0, 1]
i= 3 j= 0 F= [0, 0, 1]
i= 4 j= 0 F= [0, 0, 1, 0]
i= 5 j= 1 F= [0, 0, 1, 0, 1]
i= 6 j= 2 F= [0, 0, 1, 0, 1, 2]
i= 7 j= 3 F= [0, 0, 1, 0, 1, 2, 3]
```

Q13 -

- Avant de répondre à cette question, il est important de comprendre que **`fonctionannexe(M)`** renvoie une liste F telle que, pour $1 \leq i \leq$ **`len(M)`**, **`F[i]`** est égal à la longueur du plus grand préfixe de M se terminant en **`M[i]`** (c'est-à-dire qui est aussi un suffixe de **`M[:i+1]`**).
  En effet, dans la procédure, l'indice **`i`** sert à parcourir la liste M et l'indice **`j`** sert à parcourir le début de la liste ; tant que **`M[i]`** coïncide avec une lettre du préfixe en cours, **`i`** et **`j`** augmentent, et la longueur du préfixe trouvé est stockée dans **`F`** ; dès qu'il n'y a plus coïncidence, soit $j = 0$, auquel cas le caractère **`M[i]`** ne peut pas correspondre à un préfixe, donc **`F[i]`** vaudra 0, soit $j > 0$, auquel cas, puisque tous les caractères entre **`M[i-j]`** et **`M[i-1]`** coïncidaient déjà avec un préfixe de M (de longueur$j$), on va chercher si le caractère **`M[i]`** ne serait pas la fin d'un autre préfixe, forcément après les **`F[j-1]`** premiers caractères.

  Pour bien comprendre, essayez de suivre le cheminement de l'algorithme avec la chaîne de caractère :

  'AACAAACAAC'

  pour laquelle on trouve :

  F=[0, 1, 0, 1, 2, 2, 3, 4, 5, 3].

- Réponses aux questions posées :
  - La ligne 2 permet de remplir la liste F associée à la chaîne M, cette liste correspondant à ce qui est expliqué plus haut.
  - Les lignes 3 et 4 initialisent deux variables i et j à zéro. La variable i désigne un indice permettant de parcourir la chaîne T où l'on recherche le motif M ; la variable j désigne un indice permettant de parcourir la chaîne M.
  - Les lignes correspondant au cas où l'on a trouvé le mot sont la ligne 7 (l'indice j a atteint la fin du motif M) et la ligne 8, où l'on renvoie l'indice dans T où se trouve la première occurence de M.
  - Les lignes correspondant au cas où l'on a trouvé deux lettres identiques dans T (indice i) et dans M (indice j) sont les lignes 6 à 11.
    Dans ce cas, si l'on a atteint la fin du motif M, on a trouvé M dans T (voir question précédente), sinon on continue la recherche en passant au caractère suivant dans T et dans M.
  - Les lignes correspondant au cas où l'on a trouvé deux lettres différentes sont les lignes 13 à 16.
    Dans ce cas :

— soit $j = 0$: cela signifie que l'on comparait `T[i]` au 1er caractère de M ; cette comparaison a échouée, on passe donc au caractère suivant dans T (c'est-à-dire i augmente de 1).

— soit $j > 0$; dans ce cas les lettres situées en `T[i]` et `M[j]` sont différentes, mais les j qui précèdent sont identiques. Ces j lettres pourraient donc être le début d'une correspondance réussie, et ce n'est pas la peine de les revérifier. On va donc comparer ensuite `T[i]` (avec la même valeur de i) avec le caractère qui a dans M l'indice suivant du plus grand préfixe de M qui est aussi un suffixe de la sous-chaîne `M[:j]`, c'est-à-dire d'indice `F[j-1]`.

Pour bien comprendre cet algorithme, j'ai affiché ci-dessous les différentes étapes dans la recherche du motif 'ATCGATA' dans la séquence 'ATCCATCGATGATCATCGATA' .

```
def fonctionAnnexe(M):
    F = [0]
    i = 1
    j = 0
    m = len(M)
    while i < m:
        if M[i] == M[j]:
            F.append(j+1)
            i = i+1
            j = j+1
        else:
            if j > 0:
                j = F[j-1]
            else:
                F.append(0)
                i = i+1
    return F

def KMP(M, T):
    F = fonctionAnnexe(M)
    print('Algorithle KMP avec M=',M,'et T=',T,'\n','F = ', F)
    i = 0
    j = 0
    while i < len(T):
        if T[i] == M[j]:
            if j == len(M) - 1:
                return i-j  # et non return(i-j) !!
            else:
                i = i+1
                j = j+1
        else:
            if j > 0:
                j = F[j-1]
            else:
                i = i+1
        print('i=',i,' j=',j)
    return -1

print('\nRésultat:', KMP('ATCGATA','ATCCATCGATGATCATCGATA'))
```

```
Algorithle KMP avec M= ATCGATA et T= ATCCATCGATGATCATCGATA
 F =  [0, 0, 0, 0, 1, 2, 1]
i= 1  j= 1
i= 2  j= 2
i= 3  j= 3
i= 3  j= 0
i= 4  j= 0
i= 5  j= 1
i= 6  j= 2
i= 7  j= 3
i= 8  j= 4
i= 9  j= 5
i= 10  j= 6
i= 10  j= 2
i= 10  j= 0
i= 11  j= 0
i= 12  j= 1
i= 13  j= 2
i= 14  j= 3
i= 14  j= 0
i= 15  j= 1
i= 16  j= 2
i= 17  j= 3
i= 18  j= 4
i= 19  j= 5
i= 20  j= 6

Résultat: 14
```

2.4 Algorithme utilisant la structure de liste

Q14 - Je me contente ici de reproduire les programmes vus en cours ; le premier est le tri par insertion basique (celui qu'on vous demande de connaître pour les concours) ; les suivants utilisent le slicing de Python pour améliorer les performances ; le troisième utilise en plus la recherche dichotomique dans un tableau trié pour déterminer où insérer l'élément courant, et améliorer encore les performances (cf. exercice fait en TD).

```python
def TriInsertion1(L):
    """
    Tri par insertion de la liste L. La variable L est modifiée en place
    """
    for i in range(1, len(L)):
        # Invariant: L[0:i] est trié
        x = L[i]
        # insertion de x dans la sous-liste L[0:i]
        j = i-1
        while (j >= 0) and (L[j] > x):
            L[j+1] = L[j]
            j -= 1
        L[j+1] = x

def TriInsertion2(L):
    """
    Tri par insertion optimisé. Au lieu de décaler les éléments un par un,
    on décale tout un bloc en utilisant le slicing Python
    """
    for i in range(1, len(L)):
        x = L[i]
        j = i-1
        while (j >= 0) and (L[j] > x):
            j -= 1
```

```
            L[j+2:i+1] = L[j+1:i]
            L[j+1] = x

    def TriInsertion3(L):
        """
        On utilise ici la recherche dichotomique pour trouver le bon endroit
        """

        def dicho(L, x, debut, fin):
            """
            Cherche la place où insérer x dans la liste supposée triée
            entre les indices debut et fin inclus
            """
            while debut < fin:
                m = (debut + fin) // 2
                if L[m] < x:
                    debut = m + 1
                elif L[m] > x:
                    fin = m - 1
                else:
                    return m
            if L[debut] > x:
                return debut
            else:
                return debut + 1

        for i in range(1, len(L)):
            x = L[i]
            pos = dicho(L, x, 0, i-1)
            L[pos+1:i+1] = L[pos:i]
            L[pos] = x
```

Q15 - Pour trier une liste formée de chaînes de caractères, il n'y a RIEN à changer puisque les comparaisons en Python fonctionnent aussi avec les chaînes de caractères, en les comparant justement par ordre alphabétique (mais les majuscules avant les minuscules, et à condition qu'il n'y ait pas de caractère « bizarre »dans les chaînes, ce qui n'est évidemment pas le cas dans une séquence d'ADN).

Q16 -

```
    def recherchedichotomique(M, L):
        """
        la liste L est triée et on y recherche le motif M
        """
        debut = 0
        fin = len(L) - 1
        while debut < fin:
            m = (debut + fin) // 2
            if L[m] < M:
                debut = m + 1
            else:
                fin = m
        if L[debut] == M:
            return debut
        else:
            return -1
```

Complément :

Il est un peu dommage que le sujet n'aille pas jusqu'au bout de la méthode. Pour terminer la recherche d'une séquence, il faut donc commencer par chercher la liste des sous-motifs et la trier (cela peut se faire en même temps : chaque fois que l'on trouve un nouveau sous-motif, on l'insère à sa place) ; ensuite on recherche la séquence dans cette liste. Cela donne le programme suivant (qui n'était pas demandé).

```python
def recherche3(M, T):
    # création de la liste des sous-motifs de la chaîne T
    # qui ont la même longueur que M
    # si un sous-motif apparaît plusieurs fois, il n'est stocké qu'une seule fois
    # de plus, pour chaque sous-motif, on stocke l'indice où il débute dans T
    # donc chaque élément de la liste finale sera une liste de deux éléments:
    # [motif, indice]. Il a donc fallu adapter la recherche dichotomique
    # De plus on fait le tri par insertion en même temps, chaque fois
    # qu'on a trouvé un nouveau motif.

    def rechercheDichotomique2(M, L):
        """
        la liste L est triée et on y recherche le motif M
        Mais ici chaque élément L[i] est une liste telle que
        L[i][0] = motif et L[i][1] = indice où il apparaît
        """
        debut = 0
        fin = len(L) - 1
        while debut < fin:
            m = (debut + fin) // 2
            if L[m][0] < M:
                debut = m + 1
            else:
                fin = m
        if L[debut][0] == M:
            return L[debut][1]
        else:
            return -1

    m = len(M)
    motifs = [ [T[:m], 0] ] # initialisation: 1ère sous-chaîne position 0
    for i in range(1, len(T) - m + 1):
        # sous-chaîne commençant à l'indice i
        S = T[i:i+m]
        # insertion dans la liste motifs si pas présent
        debut = 0
        fin = len(motifs) - 1
        while debut < fin:
            milieu = (debut + fin) // 2
            if motifs[milieu][0] < S:
                debut = milieu + 1
            elif motifs[milieu][0] > S:
                fin = milieu - 1
            else:
                debut = fin = milieu
        if motifs[debut][0] != S:
            if motifs[debut][0] < S:
                debut += 1
            motifs.append( [] ) # on augmente la taille de la liste
            motifs[debut+1:] = motifs[debut:-1]
            motifs[debut] = [S, i]
    print(motifs) # juste pour que vous voyiez ce qui se passe!
    return rechercheDichotomique2(M, motifs)

print(recherche3('BLA', 'blablaBLA'))
print(recherche3('Blab', 'blablaBLA'))
print(recherche3('la', 'blablaBLA'))
```

```
[['BLA', 6], ['aBL', 5], ['abl', 2], ['bla', 0], ['laB', 4], ['lab', 1]]
6
[['aBLA', 5], ['abla', 2], ['blaB', 3], ['blab', 0], ['laBL', 4], ['labl', 1]]
-1
[['BL', 6], ['LA', 7], ['aB', 5], ['ab', 2], ['bl', 0], ['la', 1]]
1
```

2.5 Fonction de hachage et évaluation de polynôme

2.5.1 Fonction de hachage, algorithme de Karp-Rabin

Q17 - *L'énoncé n'est pas très clair : faut-il travailler en base 3 ou en base 4 ? Je pense que, comme le nombre de caractères possible est 4, il faut continuer à utiliser la base 4, mais le choix de la base 3 me semble acceptable compte tenu de la formulation de la question.*

La chaîne 'CCC' devient [1, 1, 1], et le nombre écrit $\overline{111}$ en base 4 vaut $1 \times 4^0 + 1 \times 4^1 + 1 \times 4^2 = 21$. Modulo 13, cela donne 8.

Voir le programme ci-après.

2.5.2 Évaluation de polynôme, algorithme de Hörner

Q18 - Il semble que le sujet demande une fonction d'évaluation assez primaire, où l'on calcule à chaque fois les puissances de b : pour calculer b^k, le sujet dit qu'il faut faire k multiplications (en fait, il existe un algorithme d'exponentiation rapide dont la complexité est en $\mathcal{O}(\ln k)$).

Cette méthode est de toutes façons horrible. J'ai appelé la fonction correspondante **`eval1`** ; sa complexité est d'après l'énoncé en $\mathcal{O}(n^2)$ où $n = \deg(P)$. Mais je propose aussi une fonction **`eval2`** tout aussi simple où l'on calcule les puissances de b au fur et à mesure, et dont la complexité est en $\mathcal{O}(n)$.

Q19 - Voir ci-dessous.

Q20 -

```python
def eval1(P, b):
    # méthode très primaire, mais cela semble être celle voulue par l'énoncé
    # complexité en sum(k-1) si k-1 mult. par boucle (1<=k<=n) soit en O(n^2)
    n = len(P) - 1 # degré
    return sum( [P[n-k] * b**k for k in range(len(P))] )

def eval2(P, b):
    # méthode plus élaborée, où les puissances de b sont calculées
    # au fur et à mesure par multiplication
    # il y a ici 2(n+1) additions (à cause du n-k) et 2(n+1) multiplications
    n = len(P) - 1 # degré
    puiss = 1 # puissances successives de b
    somme = 0
    for k in range(len(P)):
        somme += P[n-k] * puiss
        puiss *= b
    return somme

def hornerit(P, b):
    # algorithme de Hörner, avec n multiplications et n additions
    somme = P[0]
    for i in range(1, len(P)):
        somme = somme*b + P[i]
    return somme

def hornerrec(P, b):
    if len(P) == 1:
        return P[0]
    return P[-1] + b*hornerrec(P[:-1], b)

def hachage(T):
    seq = []
```

```
        codage = { 'A': 0, 'C': 1, 'G': 2, 'T': 3 } # dictionnaire
        b = len(codage)
        for car in T:
            seq.append(codage[car])
        return hornerit(seq, b) % 13

print(hachage('CCC'))
print(hachage('ACG'))
print(hachage('GAG'))
```

```
8
6
8
```

3 Collection Française de Bactéries Phytopathogènes

Q21 - La requête proposée permet d'obtenir le nombre de séquences d'ADN obtenues le 01/03/2018.

Q22 -

```
SELECT ADN FROM Sequence WHERE Gene = 'leuS'
```

Q23 -

```
SELECT Espece FROM Echantillon JOIN Sequence ON Echantillon.ADN = Sequence.ADN
WHERE Employe = 'Martin' AND Date = '10-03-2018'
```

ou bien, plus simplement :

```
SELECT Espece FROM Echantillon as e, Sequence as s
WHERE s.ADN = e.ADN AND Employe = 'Martin' AND Date = '10-03-2018'
```

Q24 -

```
SELECT Employe, COUNT(*) as Nombre FROM Sequence GROUP BY Employe
```