Master SVS **Examen – UE 34** décembre 2019

Tout équipement de calcul, programmation, communication est interdit. Le sujet comporte 2 parties notées séparément, chacune d'elles étant composée de plusieurs exercices indépendants. Lire d'abord la totalité de l'énoncé, et commencer par ce qui vous semble le plus facile. ***Si besoin, on pourra supposer certaines questions résolues.***

PARTIE 1

Exercice 1 : (**Qui suis-je ?**) Pour chacune des trois fonctions suivantes, décrivez ce qu'elles font. N'oubliez pas de parler du typage des arguments et du résultat des fonctions.

```
def inconnu1(t) :
    n=0
    for e in t:
        n = n + e
    return n
```

```
def inconnu2(t) :
    blanc = " "
    r = ""
    for e in t :
        r = r + e + blanc
    r = r[:-1]
    return r
```

```
def inconnu3(t) :
    r = ""
    for e in t :
        r = r + e
    return r
```

Exercice 2 : (**Les 1$^{\text{ères}}$ étapes de Fasta**) Le but de cet exercice est comprendre l'algorithme Fasta. Seules les parties les plus faciles seront abordées. On appellera *graine* un mot de longueur fixée à l'avance.

1. Écrivez une fonction `makedict` qui, à partir d'une séquence et d'une taille de graine, construit et retourne le dictionnaire contenant pour chaque graine présente dans la séquence, les positions auxquelles elle apparaît. Exemple : Soit la séquence `ATGATG` et des graines de taille 3. Le dictionnaire retourné doit être (sachant que les indices commencent à 0) : {`ATG :[0,3], TGA :[1] GAT :[2]`}.
2. Écrivez une fonction qui prend en argument une séquence et un dictionnaire de graines généré par la fonction précédente, et qui renvoie le nombre de graines communes entre les 2.
3. Écrivez la fonction `pseudoFasta` qui prend en argument une séquence requête `Sr` et une liste de séquences `LS` et qui retourne la séquence de la liste `LS` partageant le plus de graines avec la requête.

L'algorithme FASTA prend en fait aussi en compte les positions où sont trouvées les graines dans les séquences, mais ces aspects dépassent l'objectif de cet exercice.

Exercice 3 : (**Attribution simple de structure secondaire**) Les angles dièdres phi/psi d'une hélice alpha parfaite ont une valeur de -57 degrés et -47 degrés respectivement. Bien sûr, il est très rare que l'on trouve ces valeurs parfaites dans une protéine, par conséquent il est couramment accepté de tolérer une déviation de +/- 30 degrés sur celles-ci. Ci-dessous vous avez une liste de listes contenant les valeurs de phi/psi de la première hélice de la protéine 1TFE qui est un facteur d'élongation Ts.

```
L = [[48.6,53.4],[-124.9,156.7],[-66.2,-30.8],[-58.8,-43.1],[-73.9,-40.6],[-53.7,-37.5],[-80.6,-16.0],
    [-68.5,135.0],[-64.9,-23.5],[-66.9,-45.5],[-69.6,-41.0],[-62.7,-37.5],[-68.2,-38.3],[-61.2,-49.1],
    [-59.7,-41.1],[-63.2,-48.5],[-65.5,-38.5],[-64.1,-40.7],[-63.6,-40.8],[-66.4,-44.5],[-56.0,-52.5],
    [-55.4,-44.6],[-58.6,-44.0],[-77.5,-39.1],[-91.7,-11.9],[48.6,53.4]]
```

1. Écrivez une fonction `estEnHelice` qui prend en argument une liste de 2 nombres flottants représentant les valeurs de phi et psi et qui renvoie un booléen : vrai si les angles dièdres phi/psi ne sont pas trop loin des valeurs des hélices alpha parfaites (à 30 degrés près).
2. Écrivez une fonction `lesHelices` qui prend en argument une liste de listes contenant les valeurs de phi/psi pour une chaîne d'acides aminés et qui renvoie une liste de booléens : le premier booléen dit si le 1$^{\text{er}}$ résidu est en hélice ou non, et ainsi de suite.
3. Écrivez une fonction `pluslonguesoussuite()` qui prend en argument une liste de booléens et qui renvoie un couple de valeurs composé de 1) la longueur de la plus longue sous-liste qui ne contient que des booléens `True` consécutifs et de 2) la position de cette plus longue sous-liste de booléens `True`.
   Remarque : s'il y a plusieurs sous-listes de même longueur maximale, on renverra au choix, soit la position de première sous-liste, soit la position de la dernière. Bref, on privilégiera la solution la plus simple.
4. A l'aide des fonctions précédemment écrites, quelle est la commande à écrire pour connaître la longueur et la position d'une plus longue suite de résidus prédits en hélice alpha.

PARTIE 2

Exercice 4 : (**Occurrences approchées**) On cherche à écrire un programme qui prend en argument un brin d'ADN et une chaîne de caractères sur l'alphabet `ACGT` et qui renvoie toutes les occurrences approchées (position, nombre d'erreurs) de ce mot dans le brin en acceptant au plus k erreurs (mutations), nous ne considérons pas ici les insertions/délétions. Par exemple le mot `ACGTA` apparaît dans `TTTACGTACGTACGGAGTTT` en position 3 et 7 avec 0 erreur, en position 11 avec 1 erreur, en position 14 et 15 avec 3 erreurs. Évidement, si le nombre d'erreurs autorisées est supérieur à la longueur du mot recherché, toutes les positions seront des occurrences approchées de ce mot. C'est pour cela que nous vérifierons d'abord que le nombre d'erreurs autorisées est inférieur ou égal à la moitié de la longueur du mot recherché.

1. Écrire une fonction `nbmutations` qui prend en argument deux mots de même longueur et renvoie le nombre de mutations nécessaires pour passer de l'un à l'autre.
2. Écrire une fonction `lesOccApprochees` qui prend en argument un brin d'ADN, un mot et un entier k, et qui renvoie la liste des listes représentant les occurrences approchées du mot dans le brin d'ADN. Chaque liste de la liste est composée de 2 éléments : le 1^{er} est la position de l'occurrence et le $2^{\text{ème}}$ son nombre d'erreurs. Pour l'exemple donné dans le paragraphe introductif, `lesOccApprochees(brin,'ACGTA',3)` doit renvoyer : `[[3,0], [7,0], [11,1], [14,3], [15,3]]`.
3. Écrire une fonction `lesdiff` qui prend en entrée le mot recherché et une occurrence approchée (les 2 mots ont même longueur) et qui fournit en sortie une chaîne de caractères de même longueur composée uniquement de "=" et de "-" :
   — il y aura un "=" en position i si les lettres en position i du mot et de l'occurrence approchée sont égales,
   — il y aura un "-" en position i sinon.
4. Le **contexte préfixe** d'une occurrence approchée est défini comme suit : il s'agit des 5 lettres précédant l'occurrence si elles existent, et si l'occurrence est en position $i < 5$, alors il s'agit des i premières lettres. Le **contexte suffixe** est défini de manière similaire : il s'agit des 5 lettres succédant à l'occurrence si elles existent, et sinon, il s'agit des dernières lettres.
   Écrire les fonctions `contextPre` et `contextSuf` qui renvoient respectivement le contexte préfixe et le contexte suffixe d'une occurrence approchée (attention aux arguments à passer à ces 2 fonctions).
5. Écrire une fonction qui prend en argument un mot et un entier k et qui renvoie la chaîne de caractères obtenue par concaténation de k occurrences du mot passé en argument. **Attention :** on n'utilisera que l'opérateur de concaténation `+`, et on proscrira des expressions de la forme `mot * 5` qui renvoie la chaîne de caractères obtenue par concaténation de 5 occurrences du mot `mot`.
6. Écrire un procédure qui prend en entrée le mot recherché, la liste résultat de la fonction de la question 2 et qui affiche à l'écran 2 lignes pour chaque occurrence approchée trouvée :
   — la $1^{\text{ère}}$ ligne est composée de la concaténation du contexte préfixe, de l'occurrence approchée et du contexte suffixe, tout cela précédé par la position du contexte préfixe (voir exemple ci-dessous),
   — la $2^{\text{ème}}$ ligne a la même longueur que la $1^{\text{ère}}$ ligne, elle souligne chacun des caractères de la $1^{\text{ère}}$ ligne : pour chaque caractère qui n'est pas dans l'occurrence approchée, on souligne par un ".", sinon on souligne par un "=" ou un "-" comme à la question 3.

   Pour l'exemple donnée en préambule, et dans le cas où on s'intéresse aux occurrences approchées avec au plus 1 seule erreur, on obtiendrait :

```
0  TTTACGTACGTAC          2  TACGTACGTACGGAG        6  TACGTACGGAGTTT
   ...=====.....             .....=====.....           .....===-=....
```

Exercice 5 : (**Jeu des erreurs**) Pour chacune des 3 fonctions suivantes, trouvez les erreurs et corrigez-les.

```
def sommede1aN(n):
  res = 0
  i   = 1
  while i <= n:
    res = res + i
  return(res)
```

```
def doubleLetters(string):
  res = []
  for letter in string:
    res = res + letter*2
  return(res)
```

```
def comptage(string):
  nbOfLetters={}
  for letter in string:
    if letter in nbOfLetters:
      nbOfLetters[lettre] == nbOfLetters[lettre]+1
    else:
      nbOfLetters[lettre] == 1
  return(nbOfLetters)
```