TP4- Autour de l'ADN

Depuis la fin des années 2000, Le domaine de la génomique a subi une véritable révolution grâce à l'apparition de techniques de séquençage à haut débit (NGS : Next Generation Sequencing).

L'**annotation** est une branche de la annotation qui étudie le décryptage des séquences issues des génomes. Afin de déterminer ce qui se cache derrière une séquence d'ADN, une première approche est de chercher dans la séquence les éléments classiques présents dans un gène. Par exemple un codon initiateur et un codon stop. Un bon indice de présence d'un gène est la détermination d'un ORF (Open Reading Frame) ou cadre de lecture ouvert. Il s'agit d'une séquence comportant un codon initiateur et un codon stop séparés par un nombre raisonnable de codons codant pour des acides aminés. Un ORF trouvé dans une séquence Procaryote est un bon indice de la présence d'un gène. Chez les Eucaryote, un ORF est un indice de présence d'un exon ou d'un gène entier.

Question 1. *Récupérez sur le site cahier de prepa le fragment de séquence extrait du séquençage du génome de la bactérie Bacillus subtilis :* `Bacillus-Subtilis.txt` *ainsi que le fichier python* `adn.py` *et enregistrez-les sur votre espace* ***dans un même dossier****.*

Le fichier texte est au format **FASTA**. Il s'agit du format le plus simple et le plus couramment utilisé en génomique. Vous pouvez l'ouvrir en double-cliquant dessus pour voir comment il est construit.

Question 2. *Ouvrir le fichier* `adn.py` *et l'exécuter, il charge dans la variable* `ADN_codant` *le contenu du fichier sous forme d'une grande chaîne de caractères et l'affiche. Il y a une ligne inutile dans l'affichage qui va gêner pour la suite, modifier le script pour supprimer cette ligne*

Question 3. *Écrire une fonction* `nettoie(ch)` *qui reçoit une chaîne de caractères et renvoie la chaîne obtenue à partir de* `ch` *où seuls les caractères "*`g`*", "*`a`*", "*`t`*" et "*`c`*" ont été conservés. Par exemple :*

```
>>> nettoie("un gateau au chocolat")
'gataaccat'
```

Question 4. *Modifier le script de chargement pour nettoyer le contenu de la variable* `ADN_codant` *:*

```
>>> ADN_codant
'gagtatcttgagagacttgctccgtcccattagaatctgaatggtttttggctgc...cctctggtagagggattttgattcag'
```

On peut ensuite déduire le brin d'ADN non codant par passage au complémentaire : `a` $\longleftrightarrow$ `t` et `g` $\longleftrightarrow$ `c` :

Question 5. *En utilisant le dictionnaire suivant, qui associe les lettres entre elles,*

```
COMP={'a':'t','t':'a','c':'g','g':'c'}
```

écrire une fonction `complementaire` *qui réalise cette opération :*

```
>>> complementaire('gagtatcttgagagacttgctcc')
'ctcatagaactctctgaacgagg'
```

Pour cela,

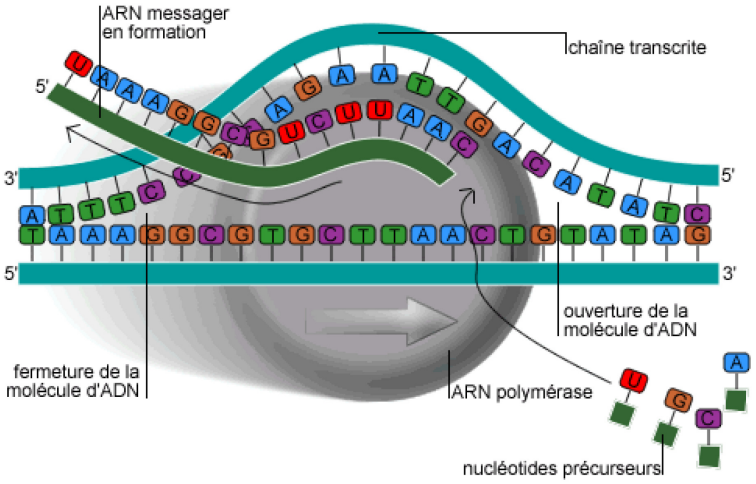
En première année, vous avez surement utilisé le logiciel en ligne **ORFfinder** :

`https://www.ncbi.nlm.nih.gov/orffinder/`

afin de rechercher différents ORF d'un brin d'ADN. Nous allons recoder quelques fonctionnalités de ce logiciel.

Commençons par la phase de transcription de **l'ADN non codant** en ARN messager : on rappelle que lors ce cette phase :

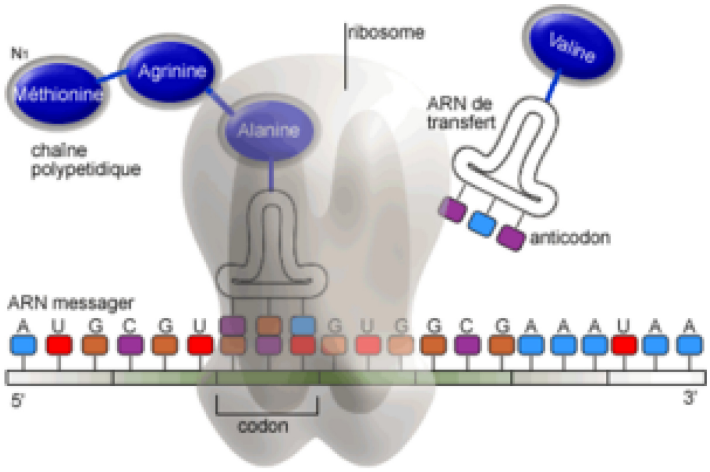
- ◇ une nucléotides d'**a**dénine de l'ADN donnera de l'**U**racile,
- ◇ une nucléotides de **g**uanine de l'ADN donnera de la **C**ytosine,
- ◇ une nucléotides de **c**ytosine de l'ADN donnera de la **G**uanine,
- ◇ une nucléotides de **t**hymine de l'ADN donnera de l'**A**dénine.



Question 6. *Écrire une fonction* `adn2arnm(adn)` *qui reçoit une chaîne représentant un brin d'ADN et revoie l'ADN messager correspondant. On utilisera un dictionnaire comme dans la question 5. Les nouvelles nucléotides seront codées en majuscules :*

```
>>> adn2arnm('ctcatagaactctctgaacgagg')
'GAGUAUCUUGAGAGACUUGCUCC'
```

La **traduction** est le processus qui va créer des **protéines** à partir de l'ARNm. La protéine est un ensemble d'**acides aminés** liés ensemble pour former une molécule utilisable par le corps humain.



L'ARN messager va être pris en charge par un ribosome qui traduit la séquence nucléotidique en acides aminés. Un acide aminé est toujours associé au même triplet de nucléotides, que l'on appelle **codon** lorsque le triplet est sur l'ARN messager.

On rappelle que l'on peut découper une chaîne de caractères ou un liste ainsi :

```
>>> chaine = "J'aime l'ADN !"
>>> chaine[9:13]
'ADN '
>>> chaine[:6]
"J'aime"
>>> chaine[7:]
"l'ADN !"
```

Question 7. *Écrire une fonction* `decoupe3(arn)` *qui reçoit une chaîne de caractères et renvoie une liste contenant la chaîne* `arn` *découpée en morceaux de longueur 3 (le dernier morceau s'il n'est pas assez long est ignoré) :*

```
>>> decoupe3('GAGUAUCUUGAGAGACUUGCUCC')
['GAG', 'UAU', 'CUU', 'GAG', 'AGA', 'CUU', 'GCU']
```

On a codé à l'aide d'un dictionnaire en Python la correspondance codon / acide aminés dans le fichier d'origine `and.py` :

```
AcidesA = {'UUU':'Phe', 'UCU':'Ser', 'UAU':'Tyr', 'UGU':'Cys', 'UUC':'Phe', 'UCC':'Ser', 'UAC':'Tyr', 'UGC':'Cys',
           'UUA':'Leu', 'UCA':'Ser', 'UAA':'Stop','UGA':'Stop','UUG':'Leu', 'UCG':'Ser', 'UAG':'Stop','UGG':'Trp',
           'CUU':'Leu', 'CCU':'Pro', 'CAU':'His', 'CGU':'Arg', 'CUC':'Leu', 'CCC':'Pro', 'CAC':'His', 'CGC':'Arg',
           'CUA':'Leu', 'CCA':'Pro', 'CAA':'Gln', 'CGA':'Arg', 'CUG':'Leu', 'CCG':'Pro', 'CAG':'Gln', 'CGG':'Arg',
           'AUU':'Ile', 'ACU':'Thr', 'AAU':'Asn', 'AGU':'Ser', 'AUC':'Ile', 'ACC':'Thr', 'AAC':'Asn', 'AGC':'Ser',
           'AUA':'Ile', 'ACA':'Thr', 'AAA':'Lys', 'AGA':'Arg', 'AUG':'STR', 'ACG':'Thr', 'AAG':'Lys', 'AGG':'Arg',
           'GUU':'Val', 'GCU':'Ala', 'GAU':'Asp', 'GGU':'Gly', 'GUC':'Val', 'GCC':'Ala', 'GAC':'Asp', 'GGC':'Gly',
           'GUA':'Val', 'GCA':'Ala', 'GAA':'Glu', 'GGA':'Gly', 'GUG':'Val', 'GCG':'Ala', 'GAG':'Glu', 'GGG':'Gly'}
```

Le codon "AUG" est appelé **codon initiateur** et indique (parfois) le début de codage d'une protéine et les codons "UGA", "UAA" ou "UAG" appelés **codons stop** indiquent eux la fin de la séquence codante.

Question 8. *On souhaite écrire une fonction* `unORF(liste, debut)` *qui reçoit une liste de codons et une valeur de départ et cherche le premier codon START après l'indice* `debut` *puis un codon STOP plus loin et renvoie un dictionnaire avec les informations ci-dessous. Si aucun codon START n'est trouvé, un dictionnaire vide est renvoyé. Si aucun codon STOP n'est trouvé après le codon START, un dictionnaire vide est renvoyé.*

```
>>> unORF(['GAG', 'UGA', 'AUG', 'CUU', 'AUG', 'AGA', 'UAA', 'GCU'],0)
{'debut': 3, 'fin': 5, 'codons': ['CUU', 'AUG', 'AGA'], 'acide': ['Leu', 'STR', 'Arg'], 'longueur': 3}
```

1. *Écrire une fonction* `trouve_START(liste, debut)` *qui renvoie l'indice du premier codon START trouvé dans la liste de codons à partir de l'indice* `debut`*. Si aucun codon START n'est trouvé, la fonction renvoie -1.*
2. *Écrire une fonction* `trouve_STOP(liste, debut)` *qui a le même effet avec le codon STOP.*
3. *Écrire une fonction* `codons2AA` *qui reçoit une liste de codons et renvoie la liste des acides aminés correspondants.*
4. *Écrire alors la fonction* `unORF` *demandée.*

Question 9. *Utiliser alors cette fonction pour réaliser la fonction* `ORF(liste)` *qui renvoie une liste de tous les ORF possibles pour la liste de codons donnée et dont la chaîne d'acides aminés a une longueur supérieure ou égale à 24 :*

```
>>> ADN_codant = nettoie(ADN_codant)
>>> ADN_noncodant = complementaire(ADN_codant)
>>> ARNm = adn2arnm(ADN_noncodant)
>>> codons = decoupe3(ARNm)
>>> ORF(codons)
[{'debut': 180, 'fin': 597, 'codons': ... , 'longueur': 418},
 {'debut': 182, 'fin': 597, 'codons': ... , 'longueur': 416},
 {'debut': 194, 'fin': 597, 'codons': ... , 'longueur': 404},
 {'debut': 218, 'fin': 597, 'codons': ... , 'longueur': 380},
 {'debut': 261, 'fin': 597, 'codons': ... , 'longueur': 337},
 {'debut': 285, 'fin': 597, 'codons': ... , 'longueur': 313},
 {'debut': 302, 'fin': 597, 'codons': ... , 'longueur': 288},
 {'debut': 319, 'fin': 597, 'codons': ... , 'longueur': 279},
 {'debut': 326, 'fin': 597, 'codons': ... , 'longueur': 272},
 {'debut': 345, 'fin': 597, 'codons': ... , 'longueur': 253},
 {'debut': 355, 'fin': 597, 'codons': ... , 'longueur': 243},
 {'debut': 365, 'fin': 597, 'codons': ..., 'longueur': 233},
 {'debut': 370, 'fin': 597, 'codons': ..., 'longueur': 228},
 {'debut': 455, 'fin': 597, 'codons': ..., 'longueur': 143},
 {'debut': 528, 'fin': 597, 'codons': ..., 'longueur': 53},
 {'debut': 619, 'fin': 649, 'codons': ..., 'longueur': 31}]
```

Question 10. *On peut améliorer la fonction précédente pour que si deux ORF ont la même fin, on ne conserve que celle de plus grande longueur :*

```
>>> ORF(codons)
[{'debut': 180, 'fin': 597, 'codons':..., 'acide': ['Asp', 'STR', 'Asn', 'Glu', ... , 'Leu', 'Ser'], 'longueur': 418},
 {'debut': 619, 'fin': 649, 'codons':..., 'acide': ['Pro', 'Glu', 'Thr', 'Lys', ..., 'Glu', 'Val'], 'longueur': 31}]
```

En réalité, comme il s'agit un extrait d'ADN, rien de nous assure que nous avons commencé exactement sur un codage d'acide aminé. Il faut donc recommencer l'opération en retirant le 1er caractère, puis les 2 premiers (colonne Frame dans ORF Finder) :

Question 11. *Écrire une fonction* `ORFs(adn)` *qui réalise cette opération (c'est ce qui correspond aux 4 parcours + de ORF Finder) et qui renvoie donc la liste de tous les dictionnaires obtenus en ne retirant aucun caractère, en retirant le premier caractère et en retirant les deux premiers caractères :*

```
>>> ORFs(ADN_codant)
[{'debut': 180, 'fin': 597, 'codons':..., 'acide': ['Asp', 'STR', 'Asn', 'Glu', ..., 'Leu', 'Ser'], 'longueur': 418},
 {'debut': 619, 'fin': 649, 'codons':..., 'acide': ['Pro', 'Glu', 'Thr', 'Lys', ..., 'Glu', 'Val'], 'longueur': 31},
 {'debut': 281, 'fin': 305, 'codons':..., 'acide': ['Phe', 'Ser', 'Ser', 'Trp', ..., 'Pro', 'Phe'], 'longueur': 25},
 {'debut': 430, 'fin': 460, 'codons':..., 'acide': ['Ala', 'Ala', 'Pro', 'Ser', ..., 'Asn', 'Pro'], 'longueur': 31}]
```

Comme on en sait pas au final lequel des deux brins est codant, il reste à effectuer la même opération avec le second brin d'ADN qu'on lit à l'envers au lieu de prendre le complémentaire du brin.

Question 12. *Écrire une fonction* `envers(ch)` *qui reçoit une chaîne et renvoie la chaine écrite dans le sens inverse*

Question 13. *Modifier alors la fonction* `ORFs` *pour quelle trouve toutes les possibilités.*

```
>>> liste_ORF = ORFs(ADN_codant)
>>> liste_ORF
[{'debut': 180, 'fin': 597, 'codons':..., 'acide': ['Asp', 'STR', 'Asn', 'Glu', ..., 'Leu', 'Ser'], 'longueur': 418},
 {'debut': 619, 'fin': 649, 'codons':..., 'acide': ['Pro', 'Glu', 'Thr', 'Lys', ..., 'Glu', 'Val'], 'longueur': 31},
 {'debut': 281, 'fin': 305, 'codons':..., 'acide': ['Phe', 'Ser', 'Ser', 'Trp', ..., 'Pro', 'Phe'], 'longueur': 25},
 {'debut': 430, 'fin': 460, 'codons':..., 'acide': ['Ala', 'Ala', 'Pro', 'Ser', ..., 'Asn', 'Pro'], 'longueur': 31},
 {'debut': 124, 'fin': 147, 'codons':..., 'acide': ['Thr', 'Ser', 'Gln', 'Val', ..., 'Thr', 'Thr'], 'longueur': 24},
 {'debut': 237, 'fin': 269, 'codons':..., 'acide': ['Ala', 'Thr', 'Pro', 'Ile', ..., 'Ala', 'Ser'], 'longueur': 33},
 {'debut': 316, 'fin': 456, 'codons':..., 'acide': ['Ile', 'Ala', 'Gly', 'Thr', ..., 'Ser', 'Ser'], 'longueur': 141},
 {'debut': 258, 'fin': 285, 'codons':..., 'acide': ['Pro', 'Phe', 'Leu', 'Val', ..., 'Phe', 'Val'], 'longueur': 28},
 {'debut': 346, 'fin': 394, 'codons':..., 'acide': ['Leu', 'Lys', 'Ile', 'Ala', ..., 'Pro', 'Pro'], 'longueur': 49},
 {'debut': 554, 'fin': 593, 'codons':..., 'acide': ['Ser', 'Leu', 'Ser', 'His', ..., 'Ser', 'Leu'], 'longueur': 40}]
```

Question 14. *Réaliser un tri par sélection sur les dictionnaires selon la clé* `longueur`

```
>>> liste_ORF = tri(liste_ORF)
>>> liste_ORF
[{'debut': 180, 'fin': 597, 'codons':..., 'acide': ['Asp', 'STR', 'Asn', 'Glu', ..., 'Leu', 'Ser'], 'longueur': 418},
 {'debut': 316, 'fin': 456, 'codons':..., 'acide': ['Ile', 'Ala', 'Gly', 'Thr', ..., 'Ser', 'Ser'], 'longueur': 141},
 {'debut': 346, 'fin': 394, 'codons':..., 'acide': ['Leu', 'Lys', 'Ile', 'Ala', ..., 'Pro', 'Pro'], 'longueur': 49},
 {'debut': 554, 'fin': 593, 'codons':..., 'acide': ['Ser', 'Leu', 'Ser', 'His', ..., 'Ser', 'Leu'], 'longueur': 40},
 {'debut': 237, 'fin': 269, 'codons':..., 'acide': ['Ala', 'Thr', 'Pro', 'Ile', ..., 'Ala', 'Ser'], 'longueur': 33},
 {'debut': 619, 'fin': 649, 'codons':..., 'acide': ['Pro', 'Glu', 'Thr', 'Lys', ..., 'Glu', 'Val'], 'longueur': 31},
 {'debut': 430, 'fin': 460, 'codons':..., 'acide': ['Ala', 'Ala', 'Pro', 'Ser', ..., 'Asn', 'Pro'], 'longueur': 31},
 {'debut': 258, 'fin': 285, 'codons':..., 'acide': ['Pro', 'Phe', 'Leu', 'Val', ..., 'Phe', 'Val'], 'longueur': 28},
 {'debut': 281, 'fin': 305, 'codons':..., 'acide': ['Phe', 'Ser', 'Ser', 'Trp', ..., 'Pro', 'Phe'], 'longueur': 25},
 {'debut': 124, 'fin': 147, 'codons':..., 'acide': ['Thr', 'Ser', 'Gln', 'Val', ..., 'Thr', 'Thr'], 'longueur': 24}]
```

Sources

- Inspiré d'un TD de Mme Mémeteau.
- https://connect.ed-diamond.com/GNU-Linux-Magazine/glmfhs-073/la-bioinformatique-avec-biopython
- Thèse de Julien BRICHE : https://tel.archives-ouvertes.fr/tel-00479671/fr/