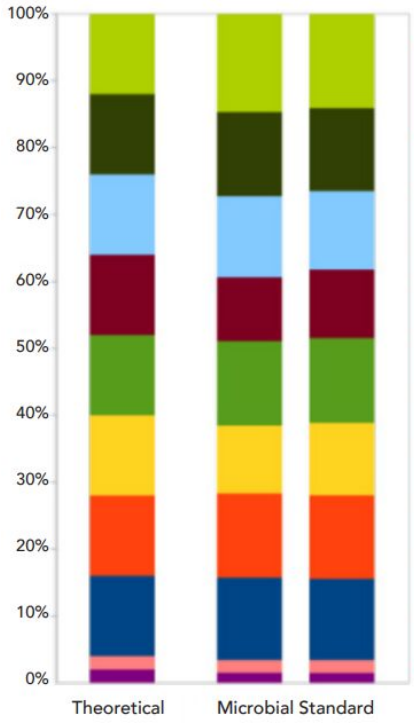
TP métagénomique 1
Développeur édition

L'objectif de ce TP sera de calculer les OTU obtenues à partir d'un séquençage "mock". Nous n'avons amplifié que les bactéries (et non les champignons). 8 espèces sont ainsi attendues.



Species	Avg. GC (%)	Gram Stain	gDNA Abun. (%)
Pseudomonas aeruinosa	66.2	-	12
Escherichia coli	56.8	-	12
Salmonella enterica	52.2	-	12
Lactobacillus fermentum	52.8	+	12
Enterococcus faecalis	37.5	+	12
Syaphylococcus aureus	32.7	+	12
Listeria monocytogenes	38.0	+	12
Bacillus subtilis	43.8	+	12
Saccharomyces cerevisiae	38.4	Yeast	2
Cryptococcus neoformans	48.2	Yeast	2

Vous devrez développer un programme python3 effectuant une dé-duplication en séquence complète ("dereplication full length"), une recherche des séquences chimériques et un regroupement basé sur un algorithme glouton ("Abundance Greedy Clustering").

Clonez le projet:

git clone https://github.com/aghozlane/agc-tp.git

Et ajoutez ce travail dans votre dépôt github (à créer en ligne):
git remote add mydepot https://github.com/mylogin/agc-tp
git push -u mydepot master

Dans le dossier agc-tp/data/, vous trouverez:

- mock_16S.fasta: séquences 16S des génomes de référence
- amplicon.fasta.gz : séquences amplifiées obtenues après trimming, clipping et merging

Dans le dossier agc-tp/agc/, vous trouverez:

- agc.py: template python du programme à réaliser
- MATCH: matrice de substitution pour des séquences nucléotidiques

Vous créerez un programme Python3 nommé agc.py dans le dossier debruijn/. Il prendra en argument:

-i, -amplicon_file fichier contenant des séquences au format FASTA
-s, *-minseqlen* Longueur minimum des séquences (optionnel - valeur par défaut 400)
-m, *-mincount* Comptage minimum des séquences (optionnel - valeur par défaut 10)
-c, *-chunk_size* Taille des partitions de séquence (optionnel - valeur par défaut 100)
-k, *-kmer_size* Longueur des "kmer" (optionnel - valeur par défaut 8)
-o, -output_file fichier de sortie avec les OTU au format FASTA

Vous utiliserez les librairies nwalign3, pytest et pylint de Python:
`pip3 install --user nwalign3 pytest pylint pytest-cov`
ou
`pip install --user nwalign3 pytest pylint pytest-cov`
selon votre installation.

La création d'un environnement conda peut résoudre les soucis d'installation.
`conda create -n agc python`
`conda activate agc`

Vous **testerez** vos fonctions à l'aide de la commande `pytest --cov=agc` à exécuter dans le dossier agc-tp/. En raison de cette contrainte, les noms des fonctions ne seront pas libre. Il sera donc impératif de respecter le nom des fonctions "imposées", de même que leur caractéristique et paramètres.

Vous vérifierez également la qualité syntaxique de votre programme en exécutant la commande: `pylint agc.py`

Chaque étape sera ponctuée de **commit** et **push** sur votre dépôt.

1. Dé-duplication en séquence "complète"

Créez un dictionnaire contenant les amplicons uniques présent dans notre échantillon de séquences. Pour cela, effectuez une lecture du fichier fasta et sélectionner les séquences uniques ayant une occurrence supérieur ou égale à ***mincount***. Deux fonctions sont ici imposées:

- **read_fasta**(amplicon_file, minseqlen)

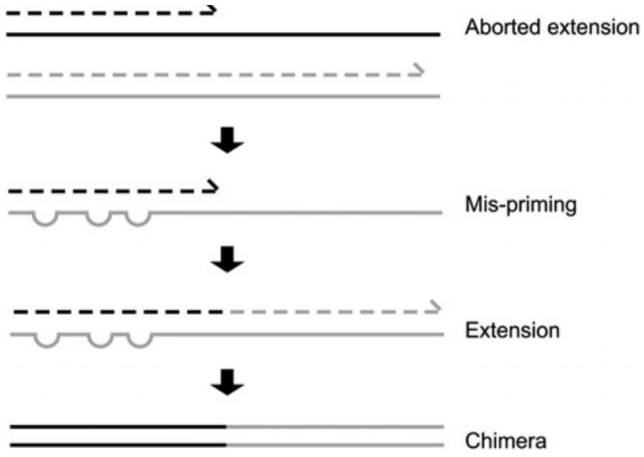
prend deux arguments correspondant au fichier fasta ou fasta.gz et à la longueur minimale des séquences et retourne un générateur de séquences de longueur l >= ***minseqlen***: ***yield*** sequence

- **dereplication_fulllength**(amplicon_file, minseqlen, mincount)

Prend trois arguments correspondant au fichier fasta, la longueur minimale des séquences et leur comptage minimum. Elle fait appel au générateur fourni par **read_fasta** et retourne un générateur des séquences uniques ayant une occurrence O>=***mincount*** ainsi que leur occurrence. Les séquences seront retournées par ordre décroissant d'occurrence: ***yield*** [sequence, count]

2. Recherche de séquences chimériques par approche "*de novo*"

Un séquence chimérique est un artefact résultant de l'amplification par PCR.
Exemple décrit par Haas et al. 2011:



Le séquençage de ces chimères est source de séquences que nous ne pourrons annoter à l'aide des banques de séquence 16S:

```
Per_id(Q,A): 94.00
---------------------------------------------------- A: S000387216
88.65                                  99.06
~~~~~~~~~~~~~~~~~~~~~~~~\/~~~~~~~~~~~~~~~~~~~~~~~~                Q:
chimera_AJ007403
~~~~~~~~~~~~~~~~~~~~~~~~/\~~~~~~~~~~~~~~~~~~~~~~~~                Q:
chimera_AJ007403
100.00                                 91.28
---------------------------------------------------- B: S000001688
Per_id(Q,B): 95.52

GGAGGCTCGTACCGCTGTCTTGTTAAGGACTGGTTTTTTACTGTCTATACAGACTCTTCA  A: S000387216
AAGACGCTTGGGTTTCACTCCTGCGCTTCGGCCGGGCCCGGCACTCGCCACAGTCTCGAG  Q: chimera_AJ007403
AAGACGCTTGGGTTTCACTCCTGCGCTTCGGCCGGGCCCGGCACTCGCCACAGTCTCGAG  B: S000001688
```

```
TACTACTGGATATCCTGATA  A: S000387216
CGTCGTCTTGATGTTCACAT  Q: chimera_AJ007403
CGTCGTCTTGATGTTCACAT  B: S000001688
```

Breakpoint

```
TGCGTTCGGATCGATTGTTGCCGTACGCTGTGTCGATTAAAGGTAATCATAAGGGCTTTC  A: S000387216
TGCGTTCGGATCGATTGTTGCCGTACGCCTGTGTCATTAAAGGTAATCATAAGGGCTTTC  Q: chimera_AJ007403
GTAACGATCGCTTCCAACCCATCCGGTGCTGTGTCGCCGGGCACGGCTTGGGAATTAACT  B: S000001688

GACTTACGACTC  A: S000387216
GACTTACGACTC  Q: chimera_AJ007403
ATTCCCAAGTCT  B: S000001688
```

Afin de détecter et supprimer ces séquences, nous effectuerons une implémentation simplifiée de l'algorithme *de novo* de recherche décrit par UCHIME [Edgar et al. 2011]:

1. Vous diviserez chaque séquence candidate en 4 segments de longueur L= ***chunk_size.***
2. Pour chaque segment, vous identifierez 8 séquences cibles présentant au minimum 1 kmer avec notre séquence candidate. Les 8 séquences cibles seront celles présentant le nombre le plus de k-mer similaires avec notre séquence candidate.
3. Vous chercherez si 2 séquences sont présentes dans chacune de ces listes. Nous appellerons ces séquences: des séquences "parentes".
4. Vous calculerez le pourcentage d'identité entre les segments des séquences parentes et de la séquence candidate.

Pour notre exemple issu de chimera slayer, nous obtenons pour des segments de 37 nt:
[
[49, 100], -> segments 1 à 37 pour S000387216 et S000001688
[51, 100], -> segments 37 à 74
[85, 59], -> segments 74 à 111
[95, 46] -> segments 111 à 148
]

5. Si l'écart type moyen des pourcentages est supérieur à **5** et que 2 segments minimum de notre séquence montrent une similarité différente à un des deux parents, nous identifierons cette séquence comme chimérique.

Fonctions imposées:

- **get_chunks**(sequence, chunk_size)

prend une séquence et un longueur de segment l: ***chunk_size*** et retourne une liste de sous-séquences de taille l non chevauchantes. A minima 4 segments doivent être obtenus par séquence.

- **cut_kmer**(sequence, kmer_size)

prend une séquence et une longueur de séquence k et retourne un générateur de tous les mots de longueur k présents dans cette séquence, ***yield*** kmer

- **get_unique_kmer**(kmer_dict, sequence, id_seq, kmer_size)

prend un dictionnaire ayant pour clé un index de kmer et pour valeur une liste d'identifiant des séquences dont ils proviennent.
Exemple:
Pour la séquence n°0 ATTCCCAAGTCT et k=8:
Kmer_dict = { "ATTCCCAA": [0], "TTCCCAAG": [0], "CCCAAGTC": [0], "CCAAGTCT": [0]}
Pour la séquence n°1 CCCAAGTCTTCC et k=8:
Kmer_dict = { "ATTCCCAA": [0], "TTCCCAAG": [0], "CCCAAGTC": [0,1], "CCAAGTCT": [0, 1], "CAAGTCTT": [1], "AAGTCTTC": [1], "AGTCTTCC": [1]}

- **search_mates**(kmer_dict, sequence, kmer_size)

prend un dictionnaire ayant pour clé un index de kmer et pour valeur une liste d'identifiant des séquences dont ils proviennent, une séquence et une longueur de kmer: kmer_size.
Vous utiliserez Counter de la librairie Collections et sa fonction most_common pour identifier les 8 séquences les plus similaires à notre séquence entrée.

```
def search_mates(kmer_dict, sequence, kmer_size):
    return [i[0] for i in Counter([ids for kmer in cut_kmer(sequence, kmer_size) if kmer in kmer_dict for ids in kmer_dict[kmer]]).most_common(8)]
```

- **get_identity**(alignment_list)

prend un alignement (sous forme de liste) et calcule le pourcentage d'identité entre les deux séquences selon la formule: $id = \frac{nb\ nucléotides\ identiques}{longueur\ de\ l'alignement}$

- **detect_chimera**(perc_identity_matrix)

prend une matrice donnant par segment le taux d'identité entre la séquence candidate et deux séquences parentes et retourne un booléen indiquant si la séquence candidate est une chimère (True) ou ne l'est pas (False). Elle suit les règles édictée au point n°5.

Si besoin:

```
def get_unique(ids):
   return {}.fromkeys(ids).keys()
```

Sinon désactivez le test

- **chimera_removal**(amplicon_file, minseqlen, mincount, chunk_size, kmer_size)

Fait appel au générateur fourni par **dereplication_fulllength** et retourne un générateur des séquences non chimérique au format: yield [sequence, count]

Faire à appel à common pour avoir les éléments commun entre 2 listes:

```
def common(lst1, lst2):
```

return list(set(lst1) & set(lst2))

3. Regroupement glouton

Vous implémenterez un regroupement glouton tel que décrit dans le cours avec la fonction:

- **abundance_greedy_clustering**(amplicon_file, minseqlen, mincount, chunk_size, kmer_size)

Fait appel à chimera removal et retourne un liste d'OTU, cette liste indiquera pour chaque séquence son occurrence (count).

- **write_OTU**(OTU_list, output_file)

Prend une liste d'OTU et le chemin vers un fichier de sortie et affiche les OTU au format:
>OTU_{numéro partant de 1} occurrence:{nombre d'occurrence à la déréplication}
{séquence au format fasta}

Vous ferez appel à la fonction fill pour respecter ce format.
Nous testerons la qualité du résultat fourni à l'aide de vsearch et des références 16S de nos séquences d'entrée: mock_16S.fasta. Alignez vos OTU contre cette banque à l'aide de la fonction usearch_global de vsearch.

4. Soumission

Connectez vous à l'adresse:
https://docs.google.com/spreadsheets/d/1K9cvTrmPwAngcCFaP_H0nNbuyjh6z7EA7oNvUWBJx1M/edit?usp=sharing

Indiquez votre dépôt et soumettez votre travail.