HMSN205
Introduction à l'algorithmique pour la bioinformatique

Responsable : **Alban MANCHERON**

L'histoire écourtée du projet :

« Mapping de données issues de nouvelles générations de séquenceurs (NGS) sur un génome de référence »

Selon **Clothilde CHENAL**

Etudiante en première année de master
Sciences et Numérique pour la Santé,
Parcours « Bioinformatique, Connaissances, Données »

Université de Montpellier

Mars 2018

Table des matières

I. Introduction

L'ADN (acide désoxyribonucléique) est le support de l'information génétique de tout être vivant. Le génome qu'il contient leur permet de fonctionner et de se reproduire.
Chacun des doubles brins de l'ADN est un polymère dont l'unité est le nucléotide. Ce dernier est constitué par :

- une base azotée (adénine (A), cytosine (C), guanine (G) ou thymine (T)),
- un sucre (le désoxyribose),
- un groupement phosphate.

Une partie de ces suites de nucléotides peut être transcrite en ARN (acide ribonucléique) puis traduite protéine. Le génome humain n'est alors qu'un enchainement de quelques trois milliards de A, C, G et T, dictant ce que nous sommes et la façon dont nous réagissons à notre environnement.

Il a donc été question de séquencer le génome humain, pour mieux nous connaitre, anticiper les maladies et pourquoi pas les traiter. Le projet *Génome Humain* commence en 1990, avec l'ambition d'établir un séquençage complet. Il aura fallu treize ans, plus de trois milliards de dollars et la coopération internationale de seize laboratoires, avec des technologies de première génération (méthode de Sanger).

Le séquençage de nouvelle génération (NGS : *Next Generation Sequencing*) apparait en 2005 avec la deuxième génération. Des dizaines de milliers de séquences sont traitées en parallèle, le génome humain est traité dans l'ordre de la journée ; on parle alors de séquençage haut-débit. Plusieurs méthodes ont été développées dont la philosophie globale reste la même :

- amplification de la séquence d'ADN par PCR (*polymerase chain reaction*),
- cycles multiples d'incorporation de nucléotide, de lavage et d'identification (*reads*).

On retiendra notamment Illumina et Roche 454.

L'objectif porté par ce projet est d'être capable de reconnaitre de courtes séquences génomiques sur un génome de référence fourni.
Pour établir cette méthode, il est d'abord nécessaire de pouvoir traiter des fichiers FASTA et FASTQ ; ce sont des formats permettant de représenter des séquences nucléiques et qui, pour FASTQ, donnent également une information sur la qualité des *reads*.
Dans un second temps, on doit pouvoir indexer le génome de référence de telle sorte que l'on puisse retrouver efficacement une séquence dedans (table des suffixes).
Finalement, il est demandé d'implémenter notre solution algorithmique de *mapping*, en utilisant les premières phases.

Deux langages de programmation permettant l'orienté objet (POO) ont été proposé :

- Java, le Mal sur Terre :
    - Bien qu'il gère les types primitifs (int, double, boolean, …) et les méthodes de classes statiques, Java est globalement purement de la programmation orientée objet, les classes y sont obligatoires.
    - Compilation du code source par un code intermédiaire utilisant la machine virtuelle Java (JVM).
    - Conversions « dangereuses » doivent être explicites.
    - Instanciation d'un objet par *new*, fournit une référence sur l'objet.
    - Initialisation par défaut des champs d'objet.
    - Héritage simple.
- C++, son papa :
    - Permet tant la programmation procédurale que la POO.
    - Compilation avec plusieurs étapes (préprocessing, création des fichiers objet, éditions des liens) qui fournit un exécutable. Utilisation possible de Makefile pour la faciliter.
    - Utilisation possible de pointeurs.
    - Instanciation d'un objet automatique (déclaration) ou dynamique (*new*).
    - Gestion mémoire infernale (*new* → doit être explicitement détruit)
    - Héritage multiple possible.
    - Puissance de calcul sur de gros jeux de données.

Issue d'un parcours de biologiste, j'ai suivi lors du premier semestre les UE d'introduction à Python (mon amour) et Java. N'ayant pas pu accrocher - du tout - avec ce dernier et sa syntaxe, et vue la publicité diffusée lors des cours, j'ai décidé de travailler en C++.

« Et là, c'est le drame » - Groland

Pleine de bonne volonté et motivée pour apprendre un nouveau langage, je mets de côté mes *a priori* sur l'orienté objet, m'achète d'occasion « Apprendre le C++ » de Claude DELANNOY pour avoir les bases, tant d'un point de vue syntaxique que logique (oui, j'aime le papier).

II. Avancement des différents travaux pratiques

Comme annoncé au début de l'UE, le projet de mapping se base grandement sur les trois séances de TP. J'ai donc décidé de m'y atteler progressivement et selon l'ordre défini.

« Commence par finir ce que tu commences ! » - Le voyage de Chihiro

---

A. Réalisation d'une API de manipulation des fichiers FASTA

J'ai commencé par me focaliser sur ce que comportais théoriquement un fichier FASTA, pour chaque séquence :

- Un entête, commençant par « > » ou « ; ».
- La suite éventuelle de l'entête, commençant par « ; ».
- La séquence nucléique ou protéique selon le standard IUPAC à une lettre, sur des lignes de 60 ou 70 caractères.

J'ai fait le choix de ne pas gérer les « - » pourtant considérés comme des gaps par le code IUPAC, de la même façon que je considère le nucléotide uracile (U) comme dégénéré afin d'optimiser plus tard la gestion de la mémoire, comme vu précédemment en cours.
Aussi, j'ai décidé de me concentrer durant ce projet sur les séquences nucléiques ; il serait tout à fait possible d'implémenter mon programme pour traiter les séquences nucléotidiques.

Code IUPAC des acides nucléiques	Signification
A	
C	
G	
T	
U	
R	A ou G
Y	C, T ou U
K	G, T ou U
M	A ou C
S	C ou G
W	A, T ou U
B	C, G, T ou U
D	A, G, T ou U
H	A, C, T ou U
V	A, C ou G
N	A, C, G, T ou U
X	AN masqué

1. La technique initiale du « dépatouillage »

« Pensez votre code avant de l'implémenter. Soyez architectes dans l'âme ».
Message reçu, je passe dans un second temps des heures complètes sur mon brouillon à mettre des boucles dans des boucles, à jouer avec une armée de booléens (un booléen pour savoir si je suis dans la séquence, si j'ai un header, si je n'ai pas de retours à la ligne, si le premier caractère de la ligne était bien un chevron, … pléthore de booléens), à calculer la taille de ma séquence avec un compteur de retour à la ligne, et autres friandises.
Cela fonctionnait sur papier et la plupart du temps dans mon *main*.

Je ciaoutais (*cout* francisé + imparfait, une insulte à l'intégralité de l'Académie française) mon entête, la ligne de départ de la séquence et finalement sa longueur, quoi qu'approximative.

Forte de ma (petite) expérience en Python, je m'étais cantonnée à une lecture ligne par ligne qui ne m'avait jamais réellement fait défaut. Aussi, la fonction *getline*() avait un côté pratique non négligeable et était facilement exploitable.

J'ai également été confrontée à un souci de traitement pour les fichiers FASTA possédant une séquence, ou la dernière séquence des fichiers multi-FASTA, que je n'avais pas résolu.

2. « Je suppose que vous avez fait comme ça ? Il ne faut pas faire comme ça »

Les brouillons partent à la poubelle, avec la fonction *getline*() et ma gestion du fichier ligne par ligne.

Première révolution : le *buffer*.
Le *buffer* me permettait de stocker temporairement de manière plus sure une suite de caractères et de mieux gérer les retours à la ligne. Il suffisait de mettre en place une boucle *if* avec un compteur de caractères et la longueur du *buffer,* où les deux étaient comparés.
Pour ce qui est de sa taille, n'ayant aucune idée de ce qui pourrait être adapté, j'ai tenté d'en discuter avec certains M2, pas beaucoup plus avancés sur la question, qui avaient utilisé 2048. J'ai finalement décidé de garder l'exemple donné en cours : 1024.

Deuxième révolution : le *switch*.
Je n'étais pas encore arrivée à la page 79/760 de mon livre.
Le *switch* m'a permis de supprimer beaucoup de booléens. La notion d'état dans le fichier simplifie et affine le traitement des caractères dans le *buffer.*
J'ai donc utilisé les états IN_HEADER et IN_SEQUENCE.

J'ai pu également résoudre mon problème de séquence en fin de fichier, en comparant la position courante dans le fichier et sa taille.

« Au début tout est pur, tout est motivant. Ensuite les erreurs commencent, les compromis... Nous créons nos propres démons. » - Iron Man

---

3. « Oui mais maintenant, il faut sortir du *main* »

J'avais jusqu'ici réussi à échapper - à tords, je sais - à la programmation orientée objet. J'ai eu énormément de mal à apprécier la scission du programme et à comprendre son intérêt. Créer toutes mes fonctions avant même d'implémenter la première me donnait un petit côté Numérobis…

« Il est où le problème ? - La porte au plafond là ! - Ça ? J'anticipe ! Si vous voulez faire un deuxième étage, paf ! Vous pouvez parce qu'il y a déjà une porte pour y accéder ! » - Astérix et Obélix : mission Cléopâtre

---

J'ai donc partitionné mon programme en trois et j'ai transféré les fonctions que j'appliquais dans le *main* à SeqFasta.cpp, en les déclarant dans SeqFasta.h. :

- SeqFasta.h
  - Attributs, constructeurs, accesseurs, mutateurs et méthodes
- SeqFasta.cpp
  - Spécification de SeqFasta.h
- main.cpp
  - Application

J'en ai profité pour mettre en place mon Makefile, de manière simple mais fonctionnelle.

```
prog: SeqFasta.o main.o
	@echo "- Création du fichier prog"
	g++ -o prog main.o SeqFasta.o -g -std=c++11
	@echo "- Fichier prog créé \n"
SeqFasta.o : SeqFasta.cpp SeqFasta.h
	@echo "- Compilation du fichier SeqFasta.cpp"
	g++ -o SeqFasta.o -c SeqFasta.cpp -g -Wall -ansi -pedantic -std=c++11
	@echo "- Fin de compilation, création de SeqFasta.o \n"
main.o: main.cpp
	@echo "- Compilation du fichier main.cpp"
	g++ -o main.o -c main.cpp -g -Wall -ansi -pedantic -std=c++11
	@echo "- Fin de compilation, création de main.o \n"
```

Makefile

4. Au moment du rendu

A l'heure actuelle, mon *parsing* de FASTA tourne et je suis capable (miracle de le vie) de faire apparaitre successivement tous les éléments de mes objets SeqFasta (header, début de séquence et longueur exacte de la séquence) sur le terminal.
Seulement, je suis confrontée à une erreur depuis quelques jours m'empêchant de créer un vecteur dans lequel je rangerais ces objets SeqFasta. Etrangement, lorsque je copie-colle la fonction dans le *main*, le programme compile. J'en déduis que l'erreur vient du lien entre le tableau créé et son appel. J'espère avoir résolu le problème d'ici la présentation.

Il est donc impossible pour l'instant de manipuler les séquences facilement (calcul des séquences complémentaires inversées, extraction de sous-séquences, …).

Aussi, pour des soucis de gestion de mémoire, il serait de bon goût d'utiliser les fichiers EncodedSequence fournis permettant de d'encoder les séquences nucléiques en binaire et d'économiser de la ressource mémoire lors du stockage.

B. Réalisation d'une API de manipulation des fichiers FASTA / FASTQ

Comme spécifié dans l'introduction, le format FASTQ est inspiré de FASTA, auquel on aurait ajouté la qualité de chaque *read.* On a donc pour chaque séquence :

- Un entête commençant par « @ ».
- La séquence sur une ou plusieurs lignes selon les normes IUPAC.
- Un séparateur commençant par « + », pouvant être suivi de la copie identique de l'entête.
- La qualité de la séquence, sur une ou plusieurs lignes, comptant autant de caractères que la séquence (une qualité par nucléotide) codé par les symboles 33 à 126 de la table ASCII.

1. Du FASTA en plus compliqué … Héritage ?

Lorsque j'ai dû travailler sur les fichiers de séquences FASTQ, j'ai tout de suite pensé à l'héritage, que nous avions pu mettre en pratique avec Java. En effet, les formats FASTA et FASTQ partagent des similitudes de par leurs entêtes ou la façon dont sont codées leurs séquences. On pourrait imaginer des méthodes partagées et adaptées.

Seulement, par manque de temps et de compétences (non pas de capacités, j'espère), je n'ai pas créé de super classe de laquelle dériverait mes classes SeqFasta et SeqFatsq.
J'ai préféré dans un premier temps dupliquer SeqFasta.cpp et SeqFasta.h pour les optimiser afin d'obtenir des SeqFastq.cpp et SeqFastq.h fonctionnels.

2. Au moment du rendu

Rien n'est encore complètement fonctionnel (youpidou). La semaine avant la présentation du code va être longue.

C. Réalisation d'une API pour la création et l'utilisation d'une table des suffixes

Nous atteignons probablement ici la limite définitive - actuelle - de mon programme, bien que la quasi-totalité de cette partie ait été codée en cours.

Je garderai cependant l'aspect théorique des tableaux de suffixes en tête. Ils ont pour objectif de faciliter les recherches de motifs, en optimisant la mémoire utilisée (*cf*. arbre des suffixes).
On peut alors distinguer deux parties :

- Le tri des suffixes,
- La recherche de motifs par dichotomie.

Un mot de longueur n comporte n suffixes. On indique également la position du début du suffixe en commençant à 0.

ex : ROUDOUDOU → n = 9

En partant d'un alphabet qu'il est possible d'ordonner, on peut réarranger ces suffixes.
Le tableau des suffixes T est constitué des positions de début des n suffixes rangés par ordre lexicographique croissant.

ex : Ici, on garde l'ordre alphabétique, obtient donc T = {6, 3, 7, 4, 1, 0, 8, 5, 2}.

0	ROUDOUDOU
1	OUDOUDOU
2	UDOUDOU
3	DOUDOU
4	OUDOU
5	UDOU
6	DOU
7	OU
8	U

6	DOU
3	DOUDOU
7	OU
4	OUDOU
1	OUDOUDOU
0	ROUDOUDOU
8	U
5	UDOU
2	UDOUDOU

Le tri des suffixes est un algorithme qui prend naïvement :

- En moyenne : *O(n log n)* comparaisons,
- Pire des cas (toutes les lettres sont identiques sauf la dernière) : *O(n)*.

Des algorithmes plus performants se sont cependant mis en place.

On utilise ensuite le tableau de suffixe comme index ; les suffixes commençant par le motif recherché se suivent. Pour déterminer dans quelle section du tableau se trouvent les positions recherchées, l'algorithme utilise la dichotomie, en comparant les premiers caractères.

Attention, ceci n'est pas un roudoudou mais un Rondoudou (cf. votre/ton Pokédex) !

III. Le code tel qu'il est

A. Comportement du *parsing* FASTA

Après avoir compilé SeqFasta.h, SeqFasta.cpp et main.cpp, on génère l'exécutable prog. Une fois lancé dans le terminal avec un fichier FASTA en argument, le *main* applique la fonction de *parsing* sur ce fichier si celui-ci existe.

On commence par déterminer la taille du fichier grâce à *seekg*() et *tellg*(). Celle-ci permettra de traiter les dernières séquences (FASTA unique ou dernière séquence d'un multi-FASTA).

Je crée et initialise ensuite les variables me permettant de mettre le *buffer* en place, ainsi que les variables stockant :

- l'objet SeqFasta vide,
- le *header* courant : premier attribut,
- le début de la séquence courante : deuxième attribut,
- sa longueur : troisième attribut.

Je crée également une variable stockant le nombre de séquences traitées.

Deux états sont déclarés : IN_HEADER et IN_SEQUENCE.
Je spécifie que l'état initial est nécessairement IN_HEADER (source d'erreur potentielle, à optimiser).

Dans une boucle *do*{} *while*(pas fin du fichier), je tâche de recharger ou d'incrémenter le buffer quand nécessaire. Aussi, j'utilise *switch*() en fonction de l'état, et rempli chaque attribut des objets SeqFasta, que je ne peux pour l'instant pas stocker, mais seulement afficher.

J'utilise deux fonctions statiques : isNucleotide et isBlank, librement inspirées de qui a été fait en TP.

- La fonction isNucleotide retourne un booléen, qui permet de reconnaitre n'importe quel nucléotide, qu'il soit dégénéré ou non, grâce à un booléen. Si le nucléotide lu dans la séquence est dégénéré, un message d'erreur s'affiche. De la même manière, si pendant la lecture, un caractère n'est pas reconnu comme nucléotide, un message d'erreur s'affiche.
  Pour l'optimisation, il serait judicieux de changer ces nucléotides soit de manière aléatoire (A, C, G ou T) ou de fixer une base de référence. En effet, nous avons en cours que le traitement de 4 nucléotides plutôt que de 5 permet d'économiser 50% de mémoire.
- La fonction isBlank retourne un booléen, déterminant si le caractère lu est un espace, un retour à la ligne ou un alinéa. De fait, elle permet de traiter une partie des fautes de mise en forme. Si l'état est IN_SEQUENCE, les caractères retournant vrai ne sont bien évidement pas comptés dans la longueur de la séquence.

Je pense à fermer le fichier en fin de manipulation avec *close*().

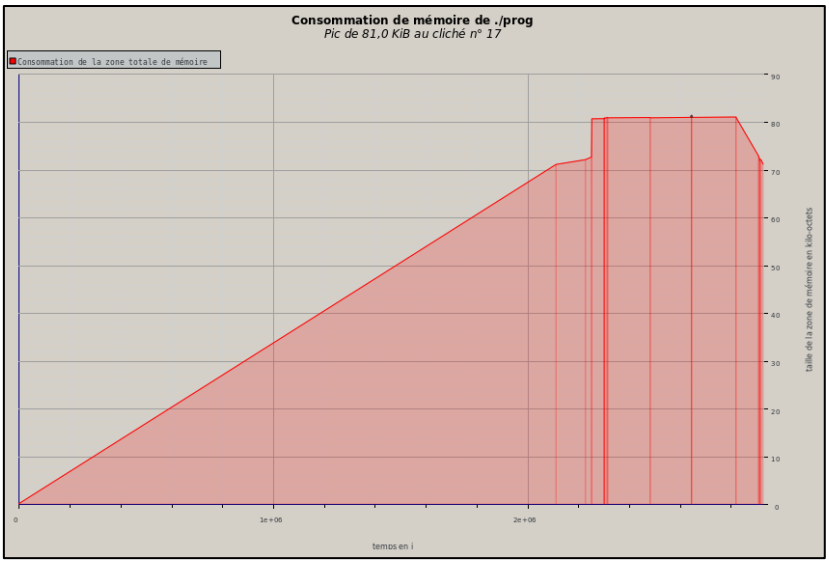
B. Notion de complexité

On peut résumer la complexité d'un algorithme à l'ordre de grandeur du nombre d'opérations à effectuer pour obtenir le résultat final. On s'intéresse à deux caractéristiques :

- la capacité en mémoire nécessaire,
- le temps d'exécution.

Si on prend n le nombre d'entrées à un programme, la complexité, en temps comme en mémoire de mon algorithme est a priori de 1*n, soit d'un ordre n, puisque je ne stocke rien et que je ne lis qu'une fois le fichier.

J'ai utilisé l'outil massif de Valgrind et son outil de visualisation pour tester mon programme malgré son avancement, afin de suivre la consommation de mémoire.
Mon fichier test est un fichier FASTA de n = 7220 caractères, contenant quatre séquences de FMO.



Graphe obtenu avec massif, représentant l'évolution de la consommation de mémoire de prog en fonction du temps.

On distingue très clairement une progression linéaire, confirmant mon hypothèse.

IV. Conclusion : mon état psychique et mes compétences avant, pendant et maintenant

Il est indéniable que mon niveau de C++ a augmenté - partir de zéro laissait une bonne marge de progression, ainsi que mes capacités en programmation orientée objet. Je suis dorénavant capable de :

- Construire une classe, avec ses attributs.
- Visualiser la séparation des données.
- Utiliser des méthodes.
- Implémenter des fonctions simples et les utiliser via un *main*.

Sont encore en cours d'acquisition (retour en primaire) :

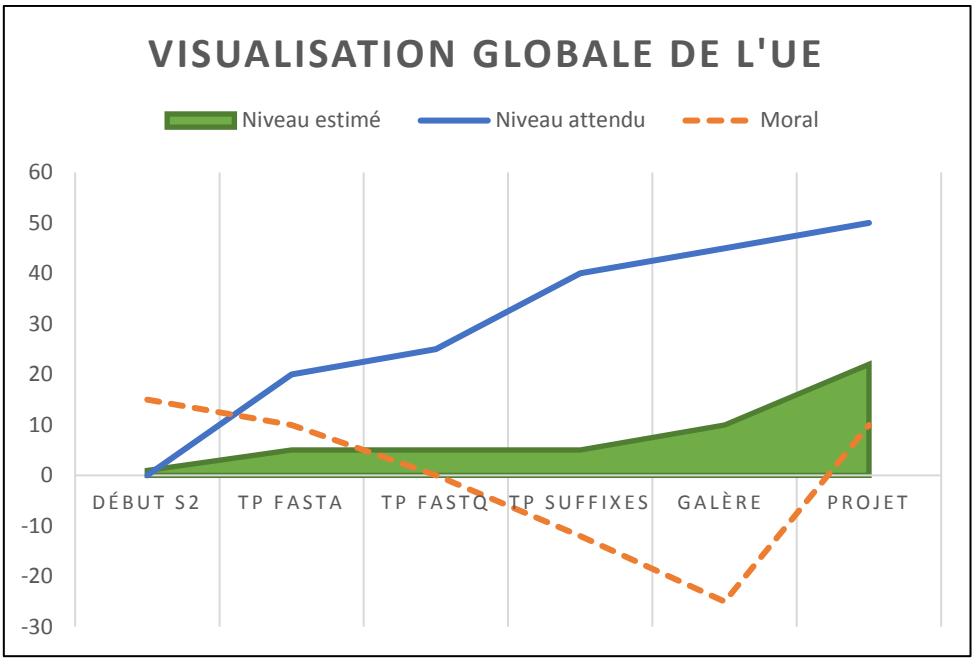
- Gestion des tableaux.
- Utilisation des masques pour la traduction en binaire.

Bien qu'encore vivante, j'ai eu beaucoup de mal à appréhender cette UE. Je préfère en général les exemples concrets à la stratégie du tâtonnement.

« Ce qui ne me tue pas me rend plus... bizarre. » - Batman : The Dark Knight

---

Après un blocage total à partir du troisième TP (où j'ai supprimé l'intégralité de mes scripts ; je vous/t'épargnerais la fameuse citation appropriée mais vulgaire de Cartman dans South Park), j'ai réussi à me reprendre en me basant sur les conseils avisés de mes camarades que je remercie.



Finalement, les bases que j'ai acquises (lire → ok, comprendre → la plupart du temps, écrire → ce n'est qu'un début) me serviront déjà pour implémenter mon code d'ici lundi prochain et probablement ultérieurement.

Et pour que mon rapport ne soit pas totalement vide de sens :

« Ce qui compte, c'est pas la force des coups que tu donnes, c'est le nombre de coups que tu encaisses tout en continuant d'avancer. » - Rocky Balboa

---