



TÉLÉCOM NANCY

RAPPORT DE PROJET TAPS

November 2020

---

# Algorithmes pour l'analyse du SARS-CoV2

---

GROUPE 13

*Étudiants :*

Mohamed Omar CHIDA  
Mathis DUMAS  
Chaima TOUNSI OMEZZINE  
Céline ZHANG

*Numéro Étudiant :*

31730598  
32012997  
32025001  
32024925

*Encadrants du projet :*

Olivier FESTOR  
Sébastien DA SILVA  
Gérald OSTER  
Titouan CARETTE



## Remerciements

Nous tenons à remercier chaque membre de l'équipe pour leurs contributions à la réalisation de ce projet. Nous tenons à remercier également Madame Sophie Mézières pour son aide concernant l'implémentation de la fonction calculant les quartiles. Nous aimeraisons remercier, de plus, Markus Piotrowski, un contributeur de la bibliothèque BioPython qui nous a aidé pour l'utilisation du module `pairwise2` et du module `PairwiseAligner`. Nous aimeraisons remercier les encadrants du projet pour nous avoir guidé durant ce projet, et notamment pour leurs réponses aux e-mails.

## Résumé

Dans le cadre de l'étude des algorithmes et de leur application, nous avons été chargés de faire des travaux de recherches sur la génomique, en particulier concernant le virus SARS-CoV-2, et de mettre en œuvre différentes méthodes (implémentations d'algorithmes, de fonctions) pour l'analyse du génome du SARS-CoV2. Ce virus est à l'origine de la pandémie mondiale 2020. Une analyse statistique des nucléotides et des acides aminés des séquences d'ARNm du génome a d'abord été réalisée. Ensuite, des algorithmes nécessaires à l'étude ont été mis en œuvre, tels que la distance de Levenshtein pour mesurer la distance minimale d'édition et l'algorithme de Needleman-Wunsch pour calculer un alignement optimal des séquences. Une poignée d'autres diverses outils ont également été mis en œuvre pour faciliter les tests et les analyses comparatives.

## Abstract

In the context of the study of algorithms and their application, we have been assigned to search and implement different methods and algorithms that will allow the analysis of the genome of SARS-CoV-2 the virus that spread widely at the beginning of 2020 starting a global pandemic. A statistical analysis of the genome nucleotides and it's amino acids was first conducted. Followed by the implementation of the tools and algorithms necessary such as the Levenshtein algorithm for measuring the minimal edit distance and Needleman-Wunsch for optimal sequence alignment. Handful of other utilities and miscellaneous tools has also been implemented to facilitate testing and benchmarking.

# Table des matières

<b>Introduction</b>	<b>4</b>
Cahier de charges . . . . .	5
Mentions légales . . . . .	5
<b>1 État de l'art</b>	<b>6</b>
1.1 Analyse descriptive d'une séquence génomique . . . . .	8
1.2 Les codons . . . . .	10
1.3 Distance de Levenshtein . . . . .	13
1.4 Algorithme de Needleman-Wunsch . . . . .	14
<b>2 Implémentation et application des algorithmes</b>	<b>18</b>
2.1 Les fonctions de description statistique . . . . .	18
2.1.1 Élaboration des fonctions statistiques . . . . .	18
2.1.2 Introduction des fonctions intermédiaires . . . . .	19
2.1.3 Applications des fonctions statistiques sur des échantillons . . . . .	21
2.2 La fonction <code>codons</code> . . . . .	24
2.2.1 Étude de l'algorithme . . . . .	26
2.2.2 Applications et interprétations . . . . .	26
2.3 La distance de Levenshtein . . . . .	29
2.3.1 <code>lev_rec</code> . . . . .	29
2.3.2 <code>lev_dp</code> . . . . .	30
2.3.3 <code>lev</code> . . . . .	31
2.3.4 Applications et interprétation . . . . .	31
2.3.5 Pistes d'amélioration possibles . . . . .	33
2.4 L'algorithme de Needleman-Wunsch . . . . .	33
2.4.1 Implémentation de <code>needleman</code> . . . . .	34
2.4.2 Implémentation de <code>needleman_all</code> . . . . .	36
2.4.3 Exécution animée de la fonction <code>needleman</code> . . . . .	37
2.4.4 Application de Needleman-Wunsch . . . . .	38
<b>3 Tests et performances</b>	<b>42</b>
3.1 Fonctions utilitaires . . . . .	42
3.2 Les tests des fonctions . . . . .	43
3.2.1 Les fonctions de statistiques . . . . .	43
3.2.2 Les fonctions codons . . . . .	44
3.2.3 Les fonctions : distance de Levenshtein . . . . .	44
3.2.4 Les fonctions : algorithme de Needleman-Wunsch . . . . .	44
3.3 Les tests de performance . . . . .	45
3.3.1 Les fonctions de statistiques . . . . .	45
3.3.2 Les fonctions codons . . . . .	49
3.3.3 Les fonctions : distance de Levenshtein . . . . .	49
3.3.4 Les fonctions : algorithme de Needleman-Wunsch . . . . .	51
<b>4 Gestion de projet</b>	<b>54</b>
4.1 Équipe de projet . . . . .	54
4.2 Analyse du projet . . . . .	54
4.2.1 Définition des objectifs . . . . .	54
4.2.2 Analyse des risques : Matrice SWOT . . . . .	55

4.3	Organisation du projet . . . . .	55
4.3.1	Durée . . . . .	55
4.3.2	Le cadre méthodologique SCRUM . . . . .	55
4.4	Outils de travail . . . . .	57
4.4.1	Communication . . . . .	57
4.4.2	IDE . . . . .	57
4.4.3	Partage du travail . . . . .	57
4.4.4	Rédaction du rapport . . . . .	58
4.5	Les réunions de projet . . . . .	58
<b>Conclusion</b>		<b>59</b>
Bilan global du projet d'équipe . . . . .		60
<b>Annexes</b>		<b>62</b>
Les déclarations sur l'honneur de non-plagiat . . . . .		62
Trello . . . . .		66
Comptes rendus de réunions . . . . .		68
Réunion d'équipe du 10 novembre 2020 . . . . .		68
Réunion d'équipe du 14 novembre 2020 . . . . .		70
Réunion d'équipe du 21 novembre 2020 . . . . .		72
Réunion d'équipe du 28 novembre 2020 . . . . .		74
Réunion d'équipe du 5 décembre 2020 . . . . .		76
Réunion d'équipe du 20 décembre 2020 . . . . .		78
Réunion d'équipe du 23 décembre 2020 . . . . .		80
Réunion d'équipe du 26 décembre 2020 . . . . .		82
Réunion d'équipe du 29 décembre 2020 . . . . .		84
Réunion d'équipe du 2 Janvier 2021 . . . . .		86
Réunion d'équipe du 4 Janvier 2021 . . . . .		88
Réunion d'équipe du 5 Janvier 2021 . . . . .		89
<b>Bibliographie</b>		<b>92</b>

## Introduction

Dans le cadre de notre module de TAPS, il nous est proposé un projet nous permettant d'acquérir et de mettre en oeuvre des compétences en gestion de projet et en programmation dynamique au travers d'un cahier des charges ayant pour finalité le développement et l'application d'algorithmes pour l'analyse de séquences génomiques du virus du SARS-CoV2 à l'origine de la pandémie de COVID-19.

L'objectif de ce projet est de réaliser des outils informatiques permettant l'étude des caractéristiques des séquences du génome SARS-CoV2, de les appliquer à des échantillons de séquences, d'étudier leur complexité théorique, de les tester sur toutes sortes de situation, et d'effectuer des tests de performances pour les comparer à la complexité théorique. Tout d'abord, nous devons faire un ensemble de fonctions qui permettra une analyse de statistique descriptive des séquences de nucléotides du génome. Ensuite, en réalisant une fonction codons, qui transformera les séquences de nucléotides en séquences d'acides aminés, nous allons faire une même analyse sur les séquences d'acides aminés. Puis, dans le but d'étudier les différences entre les séquences d'acides aminés, nous allons implémenter une fonction calculant la distance de Levenshtein, cela nous donnera la distance entre deux séquences. Enfin, pour afficher les similitudes entre deux séquences, nous réaliserons une fonction utilisant l'algorithme de Needleman-Wunsch, cette fonction permet d'afficher les alignements des séquences du génome. Pour ce faire, nous devons coder en `python`, nous avons à disposition pour les tests, les bibliothèques `pytest`, `timeit`, `biopython`, ce dernier est un outil particulier dans le domaine de la bio-informatique.

Les résultats du projet sont présentés dans ce rapport, il y a une partie consacrée à l'état de l'art, c'est-à-dire la présentation des notions en biologie, et l'explication théorique des procédés utilisés, une partie consacrée à l'implémentation et l'application de l'algorithme, elle explique le raisonnement entre les lignes de codes, et présente les résultats lors de l'exécution des codes. Dans ce rapport se trouve également une partie pour les tests et les performances des fonctions réalisées, ainsi que les comparaisons à la complexité théorique calculée dans la partie implémentation ; une partie présentant la gestion de projet de l'équipe, les moyens utilisés. À la fin de ce rapport, se trouve une partie pour les annexes, dans laquelle se trouve les déclarations de non-plagiat, les tableaux d'organisation du travail, les comptes rendus de réunions, et d'autres divers documents.

## Cahier des charges

Travaux demandés
Fonctions d'analyses statistiques
Application des fonctions d'analyses sur les séquences de nucléotides
Fonction codons
Application des fonctions d'analyses sur les séquences de codons
Fonction distance de Levenshtein
Application de la fonction distance de Levenshtein sur les séquences de codons
Fonctions algorithme Needleman-Wunsch (un seul alignement, tous les alignements)
Afficher les opérations de l'algorithme Needleman-Wunsch
Calcul de complexité
Tests des fonctions réalisées
Mesures de performances des fonctions

TABLE 1 – Le cahier des charges

## Mentions légales

Ce projet n'est pas destiné à un usage commercial, ainsi, les images présentées, notamment les images de tests, d'applications ou de gestion de projet, ne sont pas destinées à la publication.

Cependant, le caractère strictement scolaire de ce projet nous autorise à les inclure en accord avec :

- Code civil : articles 7 à 15, article 9 : respect de la vie privée
- Code pénal : articles 226-1 à 226-7 : atteinte à la vie privée
- Code de procédure civil : articles 484 à 492-1 : procédure de réfééré
- Loi n°78-17 du 6 janvier 1978 : Informatique et libertés, Article 38

Les déclarations sur l'honneur de non-plagiat des membres de l'équipe du projet sont présentés dans les quatres premières pages de l'annexe.

Cette version<sup>1</sup> du rapport a été finalisée le 5 Janvier 2021.

1. version 1

## 1 État de l'art

L'épidémie de COVID-19 [1] a débutée à Wuhan en Chine le 17 Novembre 2019. Aujourd'hui, cette pandémie a contaminé plus de 80 millions d'individus et a fait 1,75 millions de victimes. Des laboratoires du monde entier étudient de près le coronavirus SARS-CoV-2, virus vecteur de cette pandémie, dans le but de mettre au point un vaccin. Pour ce faire, les chercheurs se basent sur l'analyse du génome du virus.

Chaque être vivant possède un génome : il s'agit de l'ensemble du matériel génétique codé dans son ADN [2] ou dans son ARN [3] dans le cas de certains virus, comme pour le SARS-CoV-2 en l'occurrence. Le génome comprend notamment les gènes codant des protéines dont les organismes ont besoin pour assurer le bon fonctionnement de leur cellules. L'ADN (Acide DésoxyriboNucléique) est une macromolécule composée d'une chaîne de molécules appelés nucléotides. Les nucléotides composant l'ADN sont l'adénine (A), la guanine (G), la thymine (T) et la cytosine (C). C'est donc cette succession de nucléotides qui compose le génome et donc les gènes [4].

Pour synthétiser une protéine, la cellule transcrit d'abord le gène codant la protéine cherchée (une séquence d'ADN donc), en ARN pré-messager (ARNpm), c'est la transcription [5]. L'ARN (Acide RiboNucléique) est donc aussi une macromolécule composée d'une succession de nucléotides, à l'exception de la thymine (T) qui est remplacée par l'Uracile (U). La transcription se fait par complémentarité des nucléotides : A est associé à U, T est associé à A, G est associé à C et C est associé à G. La transcription de la séquence d'ADN "GTAC" serait donc la séquence d'ARN "CAUG" (CF fig.1).

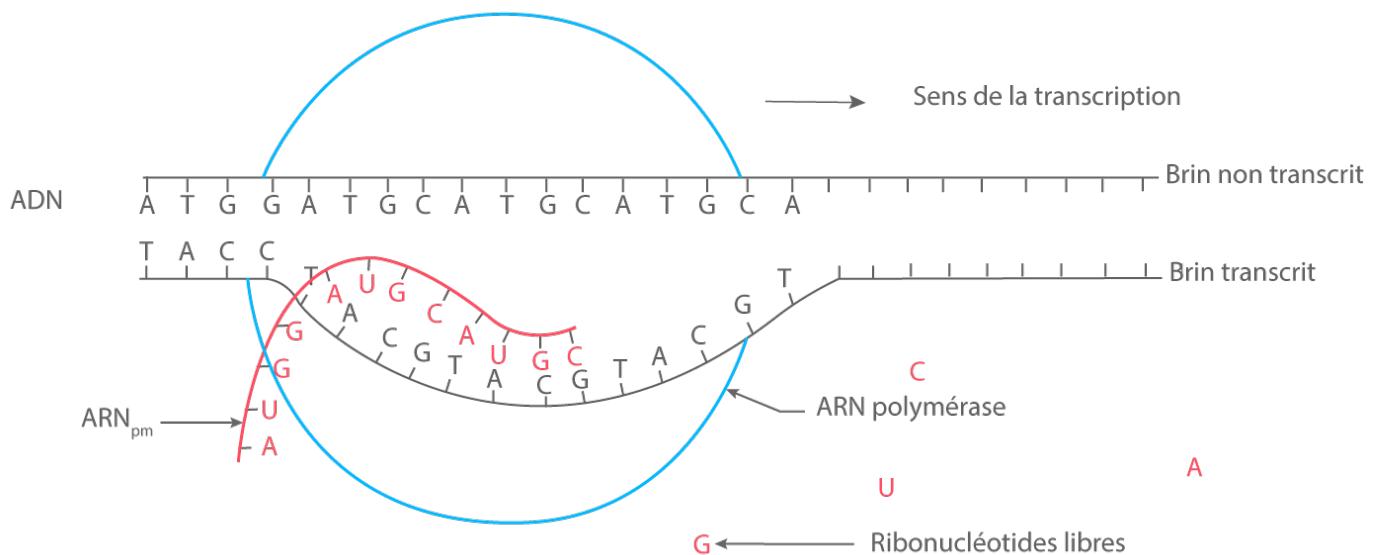


FIGURE 1 – Schéma du processus de transcription

Le brin d'ARN ainsi obtenu est appellé ARN pré-messager (ARNpm). Le gène transcrit comporte des régions codantes (exons) et des régions non-codantes (introns) pour la fabrication de la protéine. Ainsi, il est nécessaire d'enlever du brin d'ARNpm les introns pour obtenir un brin d'ARNm [6] : c'est la phase d'épissage (CF fig.2).

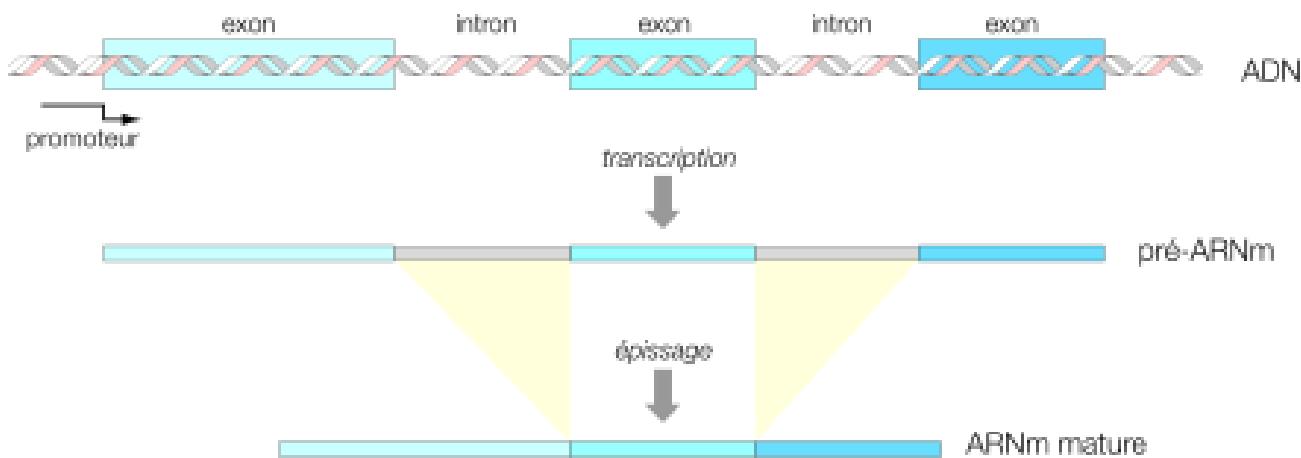


FIGURE 2 – Schéma du processus d'épissage

L'ARNm obtenu est ensuite traduit [8] en protéine par le ribosome. Une protéine est une macromolécule formée d'une succession de molécules appelées acides aminés. Le ribosome parcourt le brin d'ARNm et lit les nucléotides trois par trois. Chaque triplet de nucléotides est appelé codon. Le ribosome commence la traduction au premier codon START qu'il lit, et associe ensuite à chaque codon l'acide aminé correspondant pour former ainsi la chaîne d'acides aminés. Une fois arrivée sur un codon STOP, le ribosome se décroche et la chaîne d'acide aminé obtenue se plie sur elle-même pour former la protéine voulue (CF fig.3).

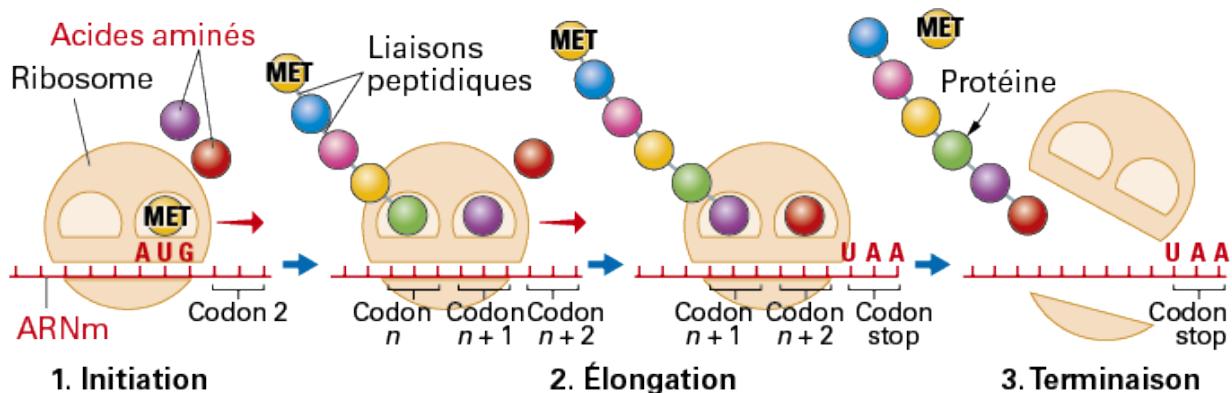


FIGURE 3 – Schéma du processus de traduction

Les virus sont des agents infectueux qui utilisent une cellule hôte pour se multiplier selon le processus de réplication [10]. Ils prennent généralement la forme de capsules contenant du matériel génétique, de l'ARN dans le cas du coronavirus SARS-CoV-2. Lors de la réplication, le virus se colle à la paroi d'une cellule hôte et fusionne avec elle, libérant ainsi son ARN dans la cellule infectée. Naturellement, les ribosomes vont traduire l'ARN présent dans la cellule, et ainsi former de nouvelles particules virales, qui seront ensuite encapsulées et libérées de la cellule infectée pour aller en infecter de nouvelles (CF fig.4).

### Cycle de reproduction du virus de la grippe

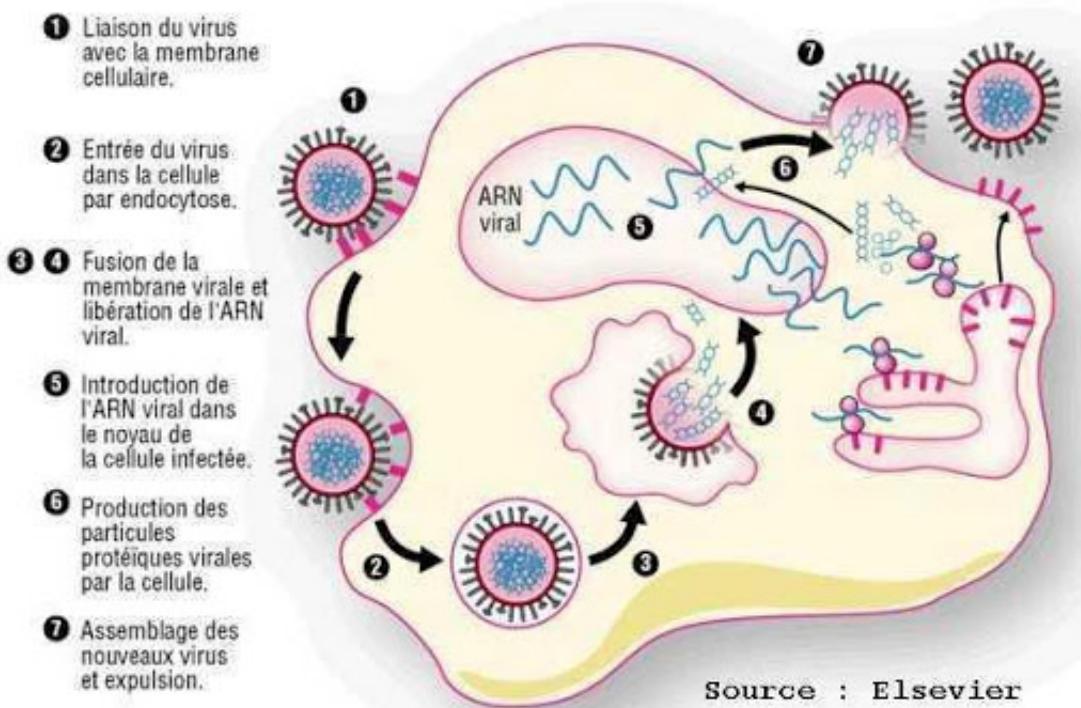


FIGURE 4 – Schéma du processus de réPLICATION

En génomique, plusieurs algorithmes ont été mis au point pour analyser, caractériser, sélectionner et comparer des séquences d'ADN. C'est ainsi que l'on peut retrouver des liens de parentés entre les différentes espèces d'animaux ou bien, dans le cadre de ce projet, ceci nous permet d'identifier les caractéristiques du génome SARS-CoV2, de comparer les différentes séquences du génome entre elles et éventuellement d'observer les différentes mutations subies par le virus SARS-CoV2.

## 1.1 Analyse descriptive d'une séquence génomique

Les analyses permettant de décrire les séquences du génome sont les fonctions de statistiques descriptives, la moyenne, la médiane, les quartiles, la variance, l'écart-type, l'intervalle interquartile et éventuellement la notion de proportion. Le but est d'appliquer ces fonctions à un échantillon (groupe de séquences d'ARNm) ceci va nous donner des valeurs qui estiment le profil général des ARNm du SARS-CoV2.

On suppose un échantillon de taille  $N$  donné, on peut appliquer l'analyse statistique sur des variables quantitatives, par exemple la taille des séquences et le nombre de chaque nucléotide des séquences de l'échantillon, ou bien sur des variables qualitatives, par exemple le nombre de nucléotides (A, U, G, C) d'une séquence (en proportion).

**La moyenne** (dite moyenne arithmétique) notée  $\bar{X}$  d'une variable discrète associé à un caractère sur une population (ou un échantillon) de taille (effectif)  $N$  est le rapport de la somme des valeurs  $X_i$  prises par cette variable pour chaque élément de la population (ou de l'échantillon) sur l'effectif total de celle-ci [12].

$$\bar{X} = \frac{X_1 + X_2 + \dots + X_N}{N} \quad (1)$$

Par exemple, la moyenne  $\bar{X}_c$  de la liste de pointures de chaussures (35, 36, 38, 39, 39, 40, 40, 42, 43) est :

$$\bar{X}_c = \frac{35 + 36 + 38 + 39 + 39 + 40 + 40 + 42 + 43}{8} = 39$$

Dans l'analyse du génome, nous appliquons cela sur la taille des séquences du génome et le nombre des nucléotides, donc la moyenne est la somme de la taille des séquences (resp. le nombre d'un nucléotide) de l'échantillon, noté  $X_i$ , sur le nombre de séquence de l'échantillon noté  $N$ .

**La proportion** notée  $P_i$  d'une caractéristique spécifique à l'étude d'un caractère est le rapport du nombre d'apparition de cette caractéristique sur l'effectif total [12]. Par exemple, la proportion d'un nucléotide dans une séquence est le nombre de nucléotide  $i$  sur la taille de la séquence noté  $n$ . En effet, la somme des proportions de tous les nucléotides vaut 1.

$$P_i = \frac{\text{nombre de nucléotides } i}{n} \quad (2)$$

Par exemple, dans une soirée, on note le nombre de personne en couple  $N_{CP}$  et célibataire  $N_S$  dans cette liste ( $N_{CP} : 8$ ,  $N_S : 12$ ), les proportions des personnes en couple  $P_{CP}$  et des personnes célibataire  $P_S$  sont la suivante :

$$P_{CP} = \frac{8}{20} = 0,4$$

et

$$P_S = \frac{12}{20} = 0,6$$

de plus

$$P_{CP} + P_S = 0,4 + 0,6 = 1$$

**La médiane** d'une variable décrivant un caractère au sein d'une population (resp. d'un échantillon) est la valeur dont 50% des éléments sur le caractère étudié lui sont inférieurs et 50% lui sont supérieurs [12]. Si plusieurs valeurs peuvent être candidates, on prend la moyenne entre la plus petite valeur et la plus grande.

En particulier, lorsque nous avons une variable quantitative discrète, nous étudions, par exemple, dans un ensemble de 5 individus, la taille en mètre des individus est donnée par la liste suivante (1,70 ; 1,58 ; 1,65 ; 1,70 ; 1,81).

Nous trions la liste, ce qui donne (1,58 ; 1,65 ; 1,70 ; 1,70 ; 1,81) et la valeur du milieu de la liste correspond à notre médiane, dans notre cas 1,70 est la médiane.

Cependant, si nous avons une liste de longueur paire, par exemple (1,58 ; 1,65 ; 1,70 ; 1,81), les valeurs 1,65 et 1,70 peuvent être candidates, ainsi nous prenons la moyenne entre le plus petit candidat et le plus grand comme médiane, dans notre cas c'est la moyenne de 1,65 et 1,70 donc 1,675 est notre médiane.

**Les quartiles** d'une variable discrète décrivant un caractère dans une population (resp. un échantillon) sont, en général, au nombre de trois [13] :  $Q_1$ ,  $Q_2$ ,  $Q_3$ .  $Q_2$  n'est autre que la médiane.  $Q_1$  est la valeur dont 25% des valeurs prises par la variable lui sont inférieures et 75% lui sont supérieures.  $Q_3$  est la valeur dont 75% des valeurs prises par la variable lui sont inférieures et 25% lui sont supérieures. Lorsqu'il y a plusieurs candidats possibles, nous choisissons le plus petit pour le premier quartile  $Q_1$  (l'interpolation par la plus petite valeur) et le plus grand pour le troisième quartile pour  $Q_3$  (l'interpolation par la plus grande valeur).

En particulier, si nous regardons par exemple le nombre de chocolats reçus à Noël parmi un groupe d'amis, on donne la liste triée (30, 90, 95, 100).

30 et 90 sont des candidats pour le premier quartile  $Q_1$ , mais nous choisissons l'interpolation par la plus petite valeur, donc  $Q_1 = 30$ .

95 et 100 sont des candidats pour le troisième quartile  $Q_3$ , mais nous choisissons l'interpolation par la plus grande valeur, donc  $Q_3 = 100$ .

**L'écart interquartile** noté  $\Delta Q$  est la différence entre le troisième et le premier quartile [14].

$$\Delta Q = Q_3 - Q_1 \quad (3)$$

Il nous permet de calculer la dispersion des valeurs d'un caractère étudié dans population (ou de l'échantillon). Comme le choix de l'interpolation pour  $Q_1$  et  $Q_3$  est la plus extrême, la valeur de la dispersion (l'intervalle interquartile) est la plus large.

Dans l'exemple précédent des chocolats, l'écart interquartile  $\Delta Q_c$  est :

$$\Delta Q_c = 100 - 30 = 70$$

**La variance et l'écart-type** sont aussi des indicateurs de dispersion [15]. La variance notée  $Var(X)$  d'une variable discrète  $X$  associée à un caractère d'une population (resp. un échantillon) de taille  $N$  est définie comme suit, où les  $X_i$  sont les valeurs que prend  $X$  et  $\bar{X}$  est sa valeur moyenne :

$$Var(X) = \frac{1}{N} \sum_{i=1}^N (X_i - \bar{X})^2 \quad (4)$$

L'écart-type noté  $\sigma$  de la variable  $X$  est la racine carré de la variance de la variable  $X$ .

$$\sigma = \sqrt{Var(X)} \quad (5)$$

Toujours dans l'exemple des chocolats (30, 90, 95, 100), calculons la variance et l'écart-type de cette liste. On note  $\bar{X}_c$  la moyenne du nombre de chocolat,  $V_c$  la variance du nombre de chocolats dans cette liste, et  $\sigma_c$  son écart-type.

$$\bar{X}_c = \frac{30 + 90 + 95 + 100}{4} = 78,75$$

$$V_c = \frac{1}{4}[(30 - 78,75)^2 + (90 - 78,75)^2 + (95 - 78,75)^2 + (100 - 78,75)^2] \approx 804,69$$

$$\sigma_c \approx 28,37$$

## 1.2 Les codons

Afin de décrire plus spécifiquement le génome, on va passer de l'analyse descriptive des échantillons d'ARNm à l'analyse descriptive des échantillons de séquences d'acides aminés. Le but est alors de générer ces séquences.

En effet, les acides aminés sont des molécules qui, combinées entre-elles, forment les protéines. Ces protéines sont produites dans la cellules par lecture du code génétique contenu dans l'ADN (les bases A, T, C, G).

D'abord, on procède à la transcription de la séquence d'ADN pour obtenir une séquence d'ARNm (les bases A, U, C, G). Ensuite, le ribosome, qui est l'usine à protéine de la cellule, se charge d'établir des liens entre la séquence d'ARNm et la séquence d'acides aminés à produire en mettant en place un système de lecture. Et enfin, on procède à la traduction en suivant les étapes [16] suivantes :

- Le ribosome lit un codon [17] (3 nucléotides successifs dans une séquence d'acide ribonucléique messager ARNm),
- On capte l'acide aminé correspondant à ce codon dans le milieu environnant, en suivant la table 2, par l'intermédiaire de l'ARN de transfert. Celui-ci est un ARN comptant une centaine de nucléotides et portant un acide aminé,

1 <sup>re</sup> base	2 <sup>e</sup> base								3 <sup>e</sup> base
	U		C		A		G		
U	UUU	F Phe	UCU	S Ser	UAU	Y Tyr	UGU	C Cys	U
	UUC	F Phe	UCC	S Ser	UAC	Y Tyr	UGC	C Cys	C
	UUA	L Leu	UCA	S Ser	UAA	Stop ocre	UGA	Stop opale / U Sec / W Trp	A
	UUG	L Leu / initiation	UCG	S Ser	UAG	Stop ambre / o Pyl	UGG	W Trp	G
C	CUU	L Leu	CCU	P Pro	CAU	H His	CGU	R Arg	U
	CUC	L Leu	CCC	P Pro	CAC	H His	CGC	R Arg	C
	CUA	L Leu	CCA	P Pro	CAA	Q Gln	CGA	R Arg	A
	CUG	L Leu / initiation	CCG	P Pro	CAG	Q Gln	CGG	R Arg	G
A	AUU	I Ile	ACU	T Thr	AAU	N Asn	AGU	S Ser	U
	AUC	I Ile	ACC	T Thr	AAC	N Asn	AGC	S Ser	C
	AUA	I Ile	ACA	T Thr	AAA	K Lys	AGA	R Arg	A
	AUG	M Met & initiation	ACG	T Thr	AAG	K Lys	AGG	R Arg	G
G	GUU	V Val	GCU	A Ala	GAU	D Asp	GGU	G Gly	U
	GUC	V Val	GCC	A Ala	GAC	D Asp	GCC	G Gly	C
	GUA	V Val	GCA	A Ala	GAA	E Glu	GGA	G Gly	A
	GUG	V Val	GCG	A Ala	GAG	E Glu	GGG	G Gly	G



TABLE 2 – Table des codons ARN

- on ajoute l'acide aminé à la chaîne d'acides aminés par la formation des liaisons péptidiques,
- le ribosome lit ensuite le codon suivant et le cycle se reproduit.

Mais, il faut bien noter que le ribosome reconnaît [18] des codons START (AUG) et des codons STOP (UAA, UAG et UGA) entre lesquels ce processus est effectués (figure 5). C'est ainsi qu'on obtient les séquences de protéine qui sont plus descriptives du génome.

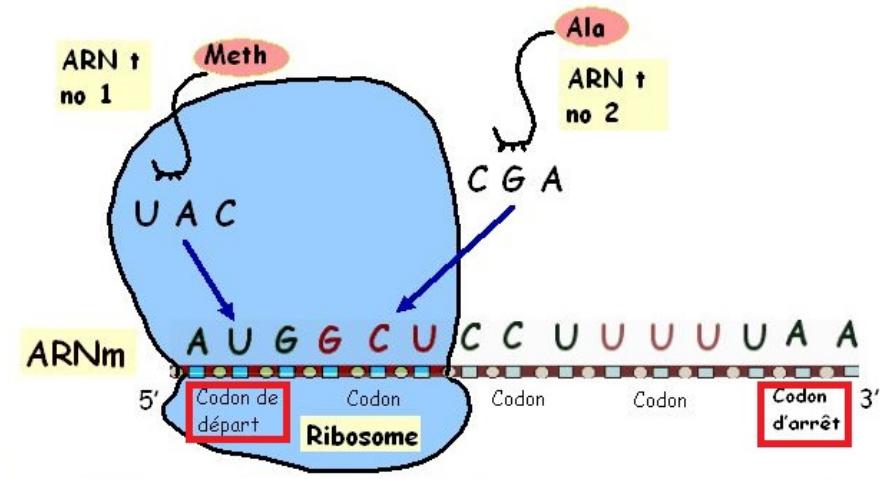


FIGURE 5 – Le processus de la traduction

De plus, pour les nucléotides, il existe plus de quatre codes IUPAC [19] de base (A, G, C , T ou U) comme indiqué sur la table 3 ce qui rend le processus plus compliqué.

Code IUPAC	Base nucléique
A	Adénine
C	Cytosine
G	Guanine
T (ou U)	Thymine (ou Uracile)
R	A ou G
Y	C ou T (U)
S	G ou C
W	A ou T (U)
K	G ou T (U)
M	A ou C
B	C ou G ou T (U)
D	A ou G ou T (U)
H	A ou C ou T (U)
V	A ou C ou G
N	N'importe quelle base
. ou -	vide

TABLE 3 – Table des codes IUPAC

### 1.3 Distance de Levenshtein

La distance de Levenshtein est une mesure mathématique de la distance entre deux mots, phrases ou ici séquences d'ARN. Concrètement, cette distance correspond au nombre minimal de lettres à ajouter, supprimer ou remplacer pour obtenir une égalité entre les deux chaînes de caractères.

Ce sont Wagner et Fischer qui, en 1974, mettent au point l'algorithme calculant cette distance de Levenshtein entre deux chaînes de caractères. Cet algorithme est un exemple de programmation dynamique. En génomique, la distance de Levenshtein peut être utilisée pour calculer un "score" de similarité entre deux séquences de nucléotides ou d'acides aminés, et ainsi identifier différents degrés de parenté entre différents génomes. D'autres applications sont envisageables, notamment en reconnaissance vocale.

L'algorithme prend en entrée deux chaînes de caractères  $seq1$  et  $seq2$  (ici "maison" et "long"), de longueur  $m$  et  $n$  (ici 6 et 4) respectivement. La phase d'initialisation consiste à créer une matrice (ici nommée "dp" pour dynamic programming) vide de dimension  $(m + 1) \times (n + 1)$  et d'attribuer à chaque case de la première ligne le numéro de la colonne correspondante et à chaque case de la première colonne le numéro de la ligne correspondante.

dp	m	0	1	2	3	4	5	6
n	\	M	A	I	S	O	N	
0								
1	L							
2	O							
3	N							
4	G							

dp	m	0	1	2	3	4	5	6
n	\	M	A	I	S	O	N	
0	0	1	2	3	4	5	6	
1	L	1						
2	O	2						
3	N	3						
4	G	4						

FIGURE 6 – Phase d'initialisation de l'algorithme de Wagner et Fischer

Ensuite, on remplit chaque case, ligne par ligne ou colonne par colonne, en respectant la formule suivante :

$$dp(i, j)_{(1,1) \leq (i,j) \leq (n,m)} = \min \begin{cases} dp(i - 1, j) + 1 \\ dp(i, j - 1) + 1 \\ dp(i - 1, j - 1) + 1 & \text{si } seq1(m) \neq seq2(n) \\ dp(i - 1, j - 1) & \text{si } seq1(m) = seq2(n) \end{cases}$$

En reprenant l'exemple, le calcul de  $dp(1, 1)$  se fait ainsi : on constate d'abord que  $seq1(1) \neq seq2(1)$  car "L" n'est pas "M". Ainsi,  $dp(1,1)$  prend la valeur minimum entre  $dp(0, 1) + 1 = 1 + 1 = 2$ ,  $dp(1, 0) + 1 = 1 + 1 = 2$  et  $dp(0, 0) + 1 = 0 + 1 = 1$ . Donc  $dp(1, 1)$  prend la valeur 1.

Concrètement, prendre la valeur du dessus correspond à effacer une lettre, prendre la valeur de gauche correspond à insérer une lettre et prendre la diagonale correspond soit à une substitution (si elles ne sont pas identiques), soit à la conservation des lettres.

dp	m	0	1	2	3	4	5	6
n	\	M	A	I	S	O	N	
0		0	1	2	3	4	5	6
1	L	1						
2	O	2						
3	N	3						
4	G	4						

dp	m	0	1	2	3	4	5	6
n	\	M	A	I	S	O	N	
0		0	1	2	3	4	5	6
1	L	1						
2	O	2						
3	N	3						
4	G	4						

FIGURE 7 – Remplissage de la matrice "dp"

Voici donc la matrice complétée de l'exemple précédent :

dp	m	0	1	2	3	4	5	6
n	\	M	A	I	S	O	N	
0		0	1	2	3	4	5	6
1	L	1						
2	O	2						
3	N	3						
4	G	4						

dp	m	0	1	2	3	4	5	6
n	\	M	A	I	S	O	N	
0		0	1	2	3	4	5	6
1	L	1	1	2	3	4	5	6
2	O	2	2	2	3	4	4	5
3	N	3	3	3	3	4	5	4
4	G	4	4	4	4	4	5	5

FIGURE 8 – Matrice "dp" complète

Finalement, on retrouve notre distance de Levenshtein dans la case  $(n, m)$  de la matrice ainsi obtenue. Dans l'exemple précédent, la distance de Levenshtein entre "maison" et "long" est donc 5.

Il est éventuellement possible d'ajouter une table des coûts en paramètre, pour modifier les coûts de suppression, d'addition ou de substitution des lettres : on peut considérer par exemple qu'une substitution coûte 2 dans le sens où elle s'apparente à une suppression puis une addition. Evidemment, les résultats obtenus ne seront pas tout à fait les mêmes, ce qui permet de personnaliser l'algorithme selon les préférences de l'utilisateur.

Par exemple, mettre un fort coût de suppression et d'addition avec un faible coût de substitution donnera une distance plus courte entre deux mots de même taille, même si leurs lettres sont différentes etc...

## 1.4 Algorithme de Needleman-Wunsch

L'algorithme de Needleman-Wunsch est un algorithme permettant de réaliser un alignement de chaînes de caractères avec un recouvrement maximal de ces dernières [20].

Cet algorithme est un exemple de programmation dynamique pour la résolution de problème d'optimisation, tout comme l'algorithme de Wagner et Fischer, auquel il est apparenté.

Son utilisation en génomique est évidente puisqu'il permet d'effectuer des alignements de séquences d'ADN ou d'ARN, qui nous permettent d'identifier les régions semblables entre deux génomes, ainsi que les lieux où ces séquences ont subies des mutations, notamment pour le génome du virus SARS-CoV2 dans le cadre de ce projet [21].

L'algorithme prend en entrée deux chaînes de caractères  $seq1$  de longueur  $n$  et  $seq2$  de longueur  $m$ , ainsi qu'un tableau de score contenant les scores à attribuer dans le cas d'un match (les deux caractères correspondent), mismatch (les deux caractères ne correspondent pas) et gap (un caractère est mis en face d'un espace). Voici ci-dessous un exemple d'alignement pour les mots "maison" et "long", ainsi que le calcul du score associé :

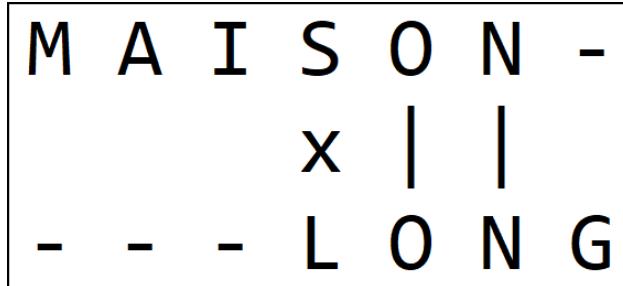


FIGURE 9 – Exemple d'un alignement entre les mots "maison" et "long", les barres représentent un match, la croix un mismatch et les tirets un gap.

En prenant un tableau de score attribuant 1 point par match,  $-1$  par gap et  $-5$  par mismatch et en l'appliquant sur l'alignement ci-dessus, on obtient un score de  $-7$ .

L'algorithme de Needleman-Wunsch se déroule en trois temps. D'abord, une phase d'initialisation, qui ressemble d'assez près à celle de l'algorithme de Wagner et Fischer : on crée une matrice de dimension  $(n + 1) \times (m + 1)$  et on attribue à chaque case de la première ligne le numéro de la colonne correspondante multiplié par le coût d'un gap et inversement pour la première colonne.

dp	m	0	1	2	3	4	5	6
n	\	M	A	I	S	O	N	-
0								
1	L							
2	O							
3	N							
4	G							

→

dp	m	0	1	2	3	4	5	6
n	\	M	A	I	S	O	N	-
0		0	-2	-4	-6	-8	-10	-12
1	L	-2						
2	O	-4						
3	N	-6						
4	G	-8						

Coûts	
Match	1
Mismatch	-3
Gap	-2

FIGURE 10 – Initialisation de l'algorithme de Needleman-Wunsch pour les mots "maison" et "long", en utilisant la table de coûts adjacente.

Ensuite, on passe à la phase de remplissage de la matrice de la même manière que pour l'algorithme de Wagner et Fischer, mais en suivant plutôt cette formule ci :

$$dp(i, j)_{(1,1) \leq (i,j) \leq (n,m)} = \max \begin{cases} dp(i - 1, j) + cout\ Gap \\ dp(i, j - 1) + cout\ Gap \\ dp(i - 1, j - 1) + cout\ Mismatch & \text{si } seq1(m) \neq seq2(n) \\ dp(i - 1, j - 1) + cout\ Match & \text{si } seq1(m) = seq2(n) \end{cases}$$

En reprenant l'exemple, le calcul de  $dp(1, 1)$  se fait ainsi : on constate d'abord que  $seq1(1) \neq seq2(1)$  car "L" n'est pas "M". Ainsi,  $dp(1,1)$  prend la valeur maximum entre  $dp(0, 1) - 2 = -2 - 2 = -4$ ,  $dp(1, 0) - 2 = -2 - 2 = -4$  et  $dp(0, 0) - 3 = 0 - 3 = -3$ .  
Donc  $dp(1, 1)$  prend la valeur  $-3$ .

On obtient finalement la matrice "dp" remplie suivante :

dp	m	0	1	2	3	4	5	6
n	\	M	A	I	S	O	N	
0		0	-2	-4	-6	-8	-10	-12
1	L	-2						
2	O	-4						
3	N	-6						
4	G	-8						

dp	m	0	1	2	3	4	5	6
n	\	M	A	I	S	O	N	
0		0	-2	-4	-6	-8	-10	-12
1	L	-2	-3	-5	-7	-9	-11	-13
2	O	-4	-5	-6	-8	-10	-8	-10
3	N	-6	-7	-8	-9	-11	-10	-7
4	G	-8	-9	-10	-11	-12	-12	-9

Coûts	
Match	1
Mismatch	-3
Gap	-2

FIGURE 11 – Matrice "dp" remplie pour les mots "maison" et "long" en utilisant la table de coûts illustrée.

Tout comme pour l'algorithme de Wagner et Fischer, on retrouve dans la case en bas à gauche le score maximal pour l'alignement des deux séquences  $seq1$  et  $seq2$ , qui est ici  $-9$ .

Finalement, on passe à la troisième et dernière phase de l'algorithme qui est le "traceback". Maintenant que nous avons rempli la matrice et déterminé le meilleur score possible pour l'alignement des deux séquences, il faut retrouver le chemin qui a permis d'arriver à ce score. Il s'agit concrètement de se positionner sur la case en bas à droite (de coordonnée  $(n, m)$ ), et de remonter la matrice jusqu'à la case en haut à gauche (de coordonnée  $(0, 0)$ ), en passant par la ou les cases utilisées pour calculer la valeur de la case dans laquelle on se trouve.

Voici, représenté par des traits rouges, l'ensemble des chemins possibles pour le traceback de la matrice "dp" :

dp	m	0	1	2	3	4	5	6
n	\	M	A	I	S	O	N	
0		0	-2	-4	-6	-8	-10	-12
1	L	-2	-3	-5	-7	-9	-11	-13
2	O	-4	-5	-6	-8	-10	-8	-10
3	N	-6	-7	-8	-9	-11	-10	-7
4	G	-8	-9	-10	-11	-12	-12	-9

FIGURE 12 – Illustration des chemins possibles à emprunter pour le traceback de la matrice "dp" pendant l'alignement des mots "maison" et "long".

Voici comment il faut "lire" le traceback de la matrice "dp" présentée ci-dessus : La case en bas à gauche,  $dp(4, 6)$ , tient sa valeur de  $dp(3, 6)$ . On effectue un déplacement vertical vers le haut, ce qui correspond à placer un gap en face du "G" de "long". La case  $dp(3, 6)$  tient sa valeur de  $dp(2, 5)$ . On effectue ici un déplacement diagonal, ce qui correspond à placer le "N" de "long" en face du "N" de "maison", c'est un match. Idem pour "O", ce qui nous mène en  $dp(1, 4)$ .

On constate ensuite que  $dp(1, 4)$  peut posséder deux origines : soit elle prend sa valeur de  $dp(0, 3)$ , ce qui correspond à mettre le "L" de "long" en face du "S" de "maison" (mismatch) ; soit elle prend sa valeur de  $dp(1, 3)$ , ce qui correspond à mettre un gap en face du "S" de "maison".

Ainsi, une fois arrivé jusqu'à  $dp(0, 0)$  l'algorithme est terminé et il renvoie deux chaînes de caractères de même longueur, composées de successions de lettres et de gaps, qui correspondent donc au meilleur (ou un des meilleurs alignements si plusieurs existent) alignement possible pour  $seq1$  et  $seq2$ , étant donné la table de score entrée en paramètre.

En reprenant l'exemple, il y a donc quatre alignements optimaux pour les mots "maison" et "long", avec un score de match de +1, un score de mismatch de -3 et un score de gap de -2 :

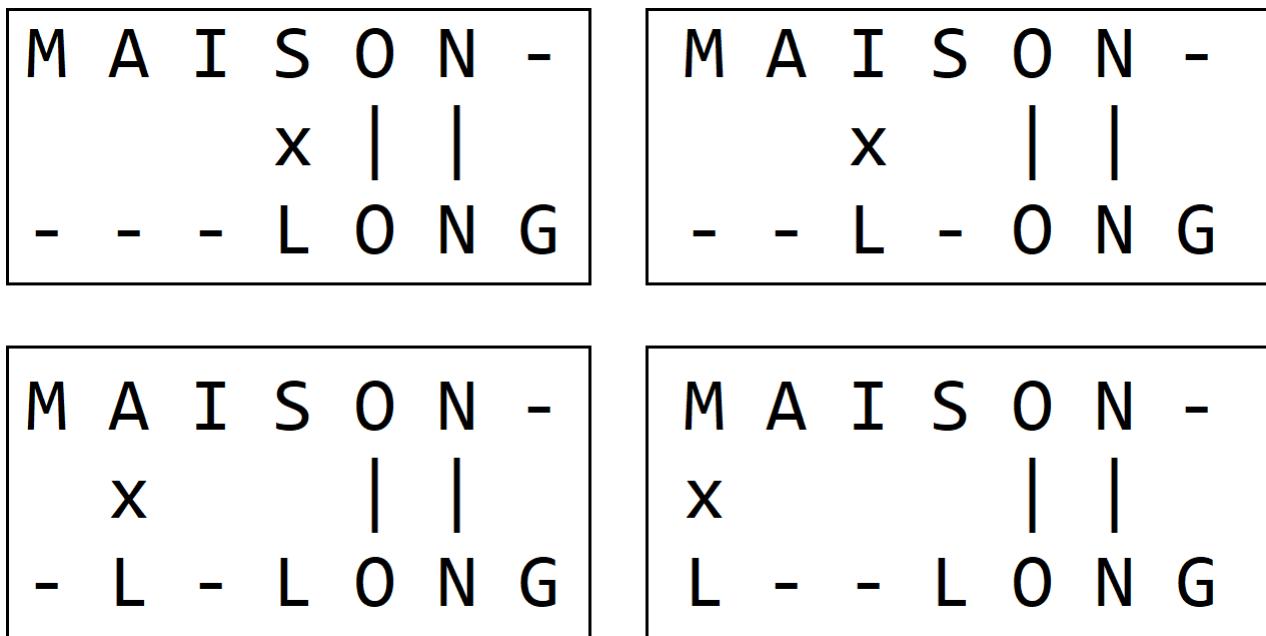


FIGURE 13 – Les quatres meilleures alignements pour les mots "maison" et "long" avec comme coûts  $\text{match}=1$ ,  $\text{mismatch}=-3$  et  $\text{gap}=-2$ .

Dans la pratique, on ajoute souvent un coût d'extension de gap, pour faire en sorte que chaque gap consécutif coûte moins cher que le premier gap de la série (-5 pour le premier gap et -1 pour tout gap consécutif suivant par exemple). Cet ajout favorise donc les grandes séquences de gap plutôt que de nombreuses petites. Ceci est essentiel pour pouvoir aligner un génome avec un gène : on aura une série de gaps au début, jusqu'à ce que la séquence du gène corresponde avec celle du génome, puis à nouveau une série de gaps jusqu'à la fin de la séquence du génome [22].

## 2 Implémentation et application des algorithmes

Dans cette partie, nous implémentons les fonctions et les algorithmes présentés dans l'état de l'art. L'objectif est d'expliquer la structure du code, les choix des données, des variables, des paramètres, de présenter les applications réalisées pour chaque partie du projet, et de donner la complexité théorique de ces codes.

### 2.1 Les fonctions de description statistique

La plupart des fonctions d'analyse statistique sont simples à implémenter, les plus grandes difficultés sont le choix des types d'interpolation pour les fonctions calculant les quartiles, le choix de la structure des fonctions qui répondent aux demandes du sujet ainsi que l'application sur des échantillons.

#### 2.1.1 Élaboration des fonctions statistiques

L'écriture des fonctions calculant les quartiles, ne peuvent pas être faites n'importe comment. Il faut choisir une interpolation<sup>2</sup> à nos fonctions puis rédiger les tests en conséquence. De ce fait, après renseignement, nous avons choisi (comme décrit dans l'état de l'art) la plus petite valeur pour le premier quartile  $Q_1$  et la plus grande pour le troisième quartile  $Q_3$ . Ceci donnera donc le plus large écart interquartile  $\Delta Q$ . Ainsi, pour les tests, on peut utiliser dans la bibliothèque `numpy`, la fonction `quantile` qui prend en entrée une liste de valeur, la valeur du fractile (dans notre cas c'est 0.25 pour  $Q_1$ , 0.75 pour  $Q_3$ ), et le type d'interpolation (`interpolation="lower"` pour  $Q_1$  et `interpolation="higher"` pour  $Q_3$ ).

Après avoir réglé le souci d'ambiguïté sur l'interpolation concernant les fonctions calculant le quartile, il est primordial de savoir comment présenter et structurer nos fonctions pour répondre aux questions posées dans la première partie, et éventuellement dans la deuxième partie qui concerne l'analyse statistique des acides aminés. Ainsi pour éviter de répéter les codes des fonctions pour chaque fonction d'application, nous avons choisi d'écrire des fonctions très générales de statistique : `moyenne`, `proportions`, `mediane`, `quartile`, `variance`, `ecart_type` et `intervalle_interquartile`. Elles prennent toutes en entrée une liste de nombres et renvoient chacune une valeur, sauf pour les fonctions `quartile` et `proportions`. `quartile` prend une liste de nombres et un entier entre 1 et 3, puis renvoie une valeur. `proportions` prend une chaîne de caractère et une base d'éléments<sup>3</sup>, et renvoie la proportion de chaque caractère dans la chaîne.

`moyenne` somme toutes les valeurs de la liste d'entrée par une boucle `for` et renvoie le résultat de la somme divisé par la taille de la liste d'entrée. Cette fonction est de complexité linéaire  $\Theta(n)$ , avec  $n$  la taille de la liste.

`proportions` fait appel à la fonction `nombre_elements` pour le décompte de chaque élément de la liste d'entrée dans une base d'entrée (NUCLEOTIDES<sup>4</sup> ou AMINO\_ACIDS<sup>5</sup>) qu'elle stock dans un dictionnaire. Cette fonction fait une boucle `for` avec des vérifications (`if`) simple dans la base d'élément, comme la taille de la base est fixe, on a une complexité linéaire  $\Theta(n)$  dans notre situation. Ensuite, s'il y a aucun élément dans la liste, la fonction retourne un dictionnaire avec tous les éléments de la

---

2. donne une unique solution  
 3. les nucléotides par exemple  
 4. est la base des nucléotides  
 5. est la base des acides aminés

base de décompte nul, sinon elle entre dans une boucle `for` et met dans un dictionnaire la division du nombre de l'élément sur l'effectif total, cela nous donne les proportions de chaque élément (nucléotides par exemple). Cette fonction est de complexité linéaire,  $\Theta(n)$ , l'appel des fonctions est en dehors de la boucle `for` et dans la boucle, il n'y a que des calculs simples.

`mediane` range la liste d'entrée de taille  $n$  avec la fonction `sorted` de python, celle-là a une complexité en  $\Theta(n \log(n))$  dans le pire des cas, et une complexité en  $\Theta(n)$  dans le meilleur des cas. La fonction `mediane` renvoie la moyenne des deux valeurs centrales de la liste triée si celle-ci est paire, sinon elle renvoie l'élément central de la liste triée. Cette fonction est de complexité quasi-linéaire,  $\Theta(n \log(n))$ .

`quartile` range la liste d'entrée de taille  $n$  avec la fonction `sorted`, si on demande le deuxième quartile, on appelle la fonction `mediane`, si la taille  $n$  de la liste est divisible par 4, on renvoie l'élément d'indice  $\frac{n}{4} - 1$  de la liste triée si le premier quartile est demandé, on renvoie celui d'indice  $\frac{3n}{4}$  si le troisième quartile est demandé ; sinon ( $n$  n'est pas multiple de 4), on renvoie l'élément d'indice  $\left\lfloor \frac{n}{4} \right\rfloor$  si le premier quartile est demandé, on renvoie celui d'indice  $\left\lfloor \frac{3n}{4} \right\rfloor$  si le troisième quartile est demandé. La complexité de cette fonction est majorée par celle du tri, donc  $\Theta(n \log(n))$ .

`variance` commence par faire la moyenne de la liste d'entrée en appelant `moyenne`, elle sauvegarde dans une variable, puis par une boucle `for` elle somme le carré de la différence de chaque élément de la liste d'entrée avec la moyenne de la liste, et enfin elle renvoie le résultat de cette somme divisé par la taille  $n$  de la liste. Dans cette fonction il y a une boucle et l'appel à la fonction `moyenne` qui est de complexité linéaire, mais leur exécution est indépendante, donc la complexité finale de `variance` est linéaire,  $\Theta(n)$ .

`ecart_type` n'est qu'un appel simple à la fonction `variance` à laquelle on applique la fonction racine carré `sqrt` de la bibliothèque `math.py` sur le résultat de l'appel. La complexité est linéaire,  $\Theta(n)$ .

`intervalle_interquartile` fait juste la différence des résultats de deux appels à la fonction `quartile` (3 et 1), donc la complexité quasi-linéaire,  $\Theta(n \log(n))$ .

Ces fonctions vont être appelées dans les fonctions d'application sur les séquences de génomes, donc il est nécessaire d'introduire d'autres fonctions pour traiter et structurer nos données.

### 2.1.2 Introduction des fonctions intermédiaires

En effet, nos fonctions très générales de statistiques citées plus haut prennent chacunes en entrée qu'une seule liste de valeurs, sauf pour la fonction `quartile`, qui prend une liste de valeurs et un nombre entier compris entre 1 et 3<sup>6</sup> qui permet de spécifier quel quartile calculer. Or, nos applications sont sur des échantillons, nous avons donc<sup>222</sup> besoin tout d'abord de convertir nos fichiers `.fasta`, dans lesquels se trouvent nos séquences, en liste de chaînes de caractères (car nos séquences du génome sont des chaînes de caractères), ceci est la raison d'être de notre fonction `fasta_to_genome`, nous donnons en entrée un fichier `.fasta` (plus précisément l'adresse du fichier), elle nous retourne

6. les cas intéressants sont 1 pour  $Q_1$  et 3 pour  $Q_3$ , 2 étant pour la médiane

une liste de chaînes de caractères qui représentent des séquences du génome, comme le présente la figure 14.

```
In[7]: fasta_to_genome("./genome/dix_minisequences.fasta")
Out[7]:
['UAAAGGUUUUAACCUUCCCAGGUACAAACCAACCAACUUUCGAUCUCUUGUAGAUCUGU',
 'UUUAUACCUUCCCAGGUACAAACCAACCAACUUUCGAUCUCUUGUAGAUCUGUUCUA',
 'UAUACCUUCCCAGGUACAAACCAACCAACUUUCGAUCUCUUGUAGAUCUGUUCUAAA',
 'AAGGUUUUAACCUUCCCAGGUACAAACCAACCAACUUUCGAUCUCUUGUAGAUCUGUUC',
 'GGUUUAUACCUUCCCAGGUACAAACCAACCAACUUUCGAUCUCUUGUAGAUCUGUUCUC',
 'UACCUUCCCAGGUACAAACCAACCAACUUUCGAUCUCUUGUAGAUCUGUUCUAAACG',
 'GUUUAUACCUUCCCAGGUACAAACCAACCAACUUUCGAUCUCUUGUAGAUCUGUUCUCU',
 'CAAACCAACCAACUUUCGAUCUCUUGUAGAUCUGUUCUCUAAACGAACUUAAAUCGU',
 'AGGUACAAACCAACCAACUUUCGAUCUCUUGUAGAUCUGUUCUCUAAACGAACUUAAA',
 'AAGGUUUUAACCUUCCCAGGUACAAACCAACUUUCGAUCUCUUGUAGAUCUGUUCUCU']
```

FIGURE 14 – Application de la fonction `fasta_to_genome`

Ensuite, il faut une fonction qui nous évalue le caractère étudié pour chaque séquence de l'échantillon. Par exemple pour la taille des séquences, nous avons la fonction `taille_ensemble` qui prend en entrée la liste de séquences et qui renvoie la liste des tailles de chaque séquence, voir la figure 15 ; pour le nombre de nucléotides 'A', 'U', 'G', ou 'C', nous avons la fonction `nombre_element_echantillon` qui prend en entrée une liste de séquences et une base d'éléments<sup>7</sup> qui calcule le nombre d'apparition de chaque élément de la base<sup>8</sup>, voir figure 16 (en faisant appelle à la fonction `nombre_elements`, celle-ci prend une chaîne de caractère en entrée et une base, et met dans un dictionnaire l'occurrence de chaque nucléotide). Le résultat de `nombre_element_echantillon` est mis dans un dictionnaire sous forme de liste de la même taille que l'échantillon pour chaque élément (nucléotide ou acide aminé).

```
In[10]: taille_ensemble(fasta_to_genome("./genome/dix_minisequences.fasta"))
Out[10]: [60, 60, 60, 60, 60, 60, 60, 60, 60]
```

FIGURE 15 – Application de la fonction `taille_ensemble`

```
In[17]: nombre_element_echantillon(fasta_to_genome("./genome/dix_minisequences.fasta"), NUCLEOTIDES)
Out[17]:
{'A': [18, 16, 18, 17, 15, 17, 15, 19, 21, 17],
 'U': [19, 21, 19, 19, 20, 18, 21, 20, 18, 19],
 'G': [8, 6, 6, 8, 8, 7, 7, 6, 7, 8],
 'C': [15, 17, 17, 16, 17, 18, 17, 15, 14, 16]}
```

FIGURE 16 – Application de la fonction `nombre_element_echantillon`

De cette manière, sur les résultats de ces fonctions (éventuellement, en sortant la liste des dictionnaires), on peut appliquer nos fonctions générales d'analyse statistique. Par exemple la fonction `moyenne` comme le montre la figure 17.

Cependant pour alléger nos codes<sup>9</sup> et réduire l'écriture d'appel à nos fonctions, nous avons introduit une fonction `call_stats` qui prend en entrée la fonction statistique que nous voulons appliquer,

7. dans notre cas ce sont les nucléotides ou les acides aminés

8. nos nucléotides ou acides aminés

9. afin d'éviter des structures répétitives

```
In[28]: moyenne(taille_ensemble(fasta_to_genome("./genome/dix_minisequences.fasta")))
Out[28]: 60.0
In[29]: moyenne(nombre_element_echantillon(fasta_to_genome("./genome/dix_minisequences.fasta"), NUCLEOTIDES)[['A']])
Out[29]: 17.3
```

FIGURE 17 – Application de la fonction moyenne

l'échantillon<sup>10</sup> à laquelle nous voulons appliquer la fonction statistique, la base sur laquelle nous évaluons (NUCLEOTIDES<sup>11</sup>, AMINO\_ACIDS<sup>12</sup>), et éventuellement un argument supplémentaire<sup>13</sup>. La fonction `call_stats` utilise les fonctions `nombre_element_echantillon` et `call_stats_on_echantillon` qui font respectivement le travail de comptage<sup>14</sup> et d'application de la fonction d'analyse statistique qui prend chaque élément du dictionnaire pour lui appliquer la fonction statistique donnée en entrée, la figure 18 montre une application de la fonction `call_stats`.

```
In[31]: call_stat(moyenne, fasta_to_genome("./genome/dix_minisequences.fasta"), NUCLEOTIDES)
Out[31]: {'A': 17.3, 'U': 19.4, 'G': 7.1, 'C': 16.2}
```

FIGURE 18 – Application de la fonction `call_stats`

La même fonction pour les statistiques sur la taille du génome a été écrite, elle s'appelle `call_stats_taille_genome`, celle-ci prend en entrée une fonction statistique, une liste de séquences et éventuellement un argument supplémentaire, elle appelle la fonction `taille_ensemble` pour donner dans une liste la taille de chaque génome de l'échantillon et c'est sur cette liste qu'elle applique la fonction statistique, la figure 19 montre ceci.

```
In[33]: call_stat_taille_genome(moyenne, fasta_to_genome("./genome/dix_minisequences.fasta"))
Out[33]: 60.0
```

FIGURE 19 – Application de la fonction `call_stats_taille_genome`

### 2.1.3 Applications des fonctions statistiques sur des échantillons

Toutefois, les fonctions `perform_all_stats` et `perform_all_stats_taille` ont été introduites pour appeler toutes les fonctions d'analyse statistique en une fois, cela réduit les lignes de codes à écrire pour les appels et répond aux premier objectif du projet qui est de décrire en utilisant la statistique descriptive le génome du SARS-CoV2.

La fonction `perform_all_stats` prend en entrée une liste de séquences et une base d'élément (nucléotides par exemple), elle appelle toutes les fonctions statistiques<sup>15</sup> et renvoie dans un dictionnaire la valeur statistique de chaque élément. Sur la figure 20, les résultats de deux applications de la fonction `perform_all_stats` sur deux échantillons de dix mille séquences de nucléotides de période différente<sup>16</sup> sont mis en évidence.

- 
- 10. la liste de séquences du génome
  - 11. est la base des nucléotides
  - 12. est la base des acides aminés
  - 13. pour le calcul de quartile
  - 14. en sauvegardant la liste de valeurs dans un dictionnaire comme vu en figure 16
  - 15. en utilisant `nombre_element_echantillon` et `call_stat_on_echantillon`
  - 16. l'une sont des séquences de janvier 2020 à août 2020, et l'autre sont des séquences d'octobre à novembre 2020

```
In[8]: perform_all_stats(fasta_to_genome("./genome/10000_sequences.fasta"), NUCLEOTIDES)
Out[8]:
{'moy': {'A': 8901.23, 'U': 9580.41, 'G': 5848.80, 'C': 5473.95},
'med': {'A': 8901.0, 'U': 9585.0, 'G': 5850.0, 'C': 5475.0},
'ecartt': {'A': 20.17, 'U': 20.03, 'G': 11.44, 'C': 11.60},
'var': {'A': 406.79, 'U': 401.07, 'G': 130.94, 'C': 134.66},
'quart1': {'A': 8894.0, 'U': 9572.0, 'G': 5848.0, 'C': 5470.0},
'quart3': {'A': 8906.0, 'U': 9589.0, 'G': 5852.0, 'C': 5478.0},
'int_quart': {'A': 12.0, 'U': 17.0, 'G': 4.0, 'C': 8.0}}

In[9]: perform_all_stats(fasta_to_genome("./genome/10000_sequences_janvier_aout_2020.fasta"), NUCLEOTIDES)
Out[9]:
{'moy': {'A': 8902.20, 'U': 9574.05, 'G': 5846.59, 'C': 5471.55},
'med': {'A': 8913.0, 'U': 9590.0, 'G': 5857.0, 'C': 5482.0},
'ecartt': {'A': 41.61, 'U': 39.67, 'G': 28.65, 'C': 28.37},
'var': {'A': 1731.43, 'U': 1574.01, 'G': 820.90, 'C': 804.63},
'quart1': {'A': 8893, 'U': 9571, 'G': 5849, 'C': 5470},
'quart3': {'A': 8922, 'U': 9596, 'G': 5861, 'C': 5488},
'int_quart': {'A': 29, 'U': 25, 'G': 12, 'C': 18}}
```

FIGURE 20 – Application de la fonction `perform_all_stats` sur deux échantillons de 10000 séquences

Les résultats des deux applications sur les deux échantillons diffèrent très peu. En effet, nous avons à peu près en moyenne 8901 (avec plus ou moins 20 d'écart) nucléotides A, 9580 (avec plus ou moins 20 d'écart) nucléotides U, 5849 (avec plus ou moins 11 d'écart) nucléotides G et 5474 (avec plus ou moins 11 d'écart) nucléotides C. Les médianes sont très proches des moyennes, ceci nous laisse croire que la répartition est plutôt symétrique, l'écart interquartile est assez petit, donc nous supposons que les valeurs sont assez regroupées autour de la médiane. De plus l'écart-type de chaque nucléotide est acceptable, il ne dépasse pas 20 pour l'échantillon d'octobre à novembre, mais celui de janvier à août est plus important (environ le double). Cette différence peut s'expliquer par le fait que l'échantillon de janvier à août est d'une période plus large que celui d'octobre à novembre, le virus a pu muter beaucoup plus de fois, produisant des séquences qui s'écartent de plus en plus de la moyenne. Un exemple graphique sur le nucléotide G est illustré par la figure 21, et un exemple d'application de la fonction `proportions` est montré sur la figure 22.

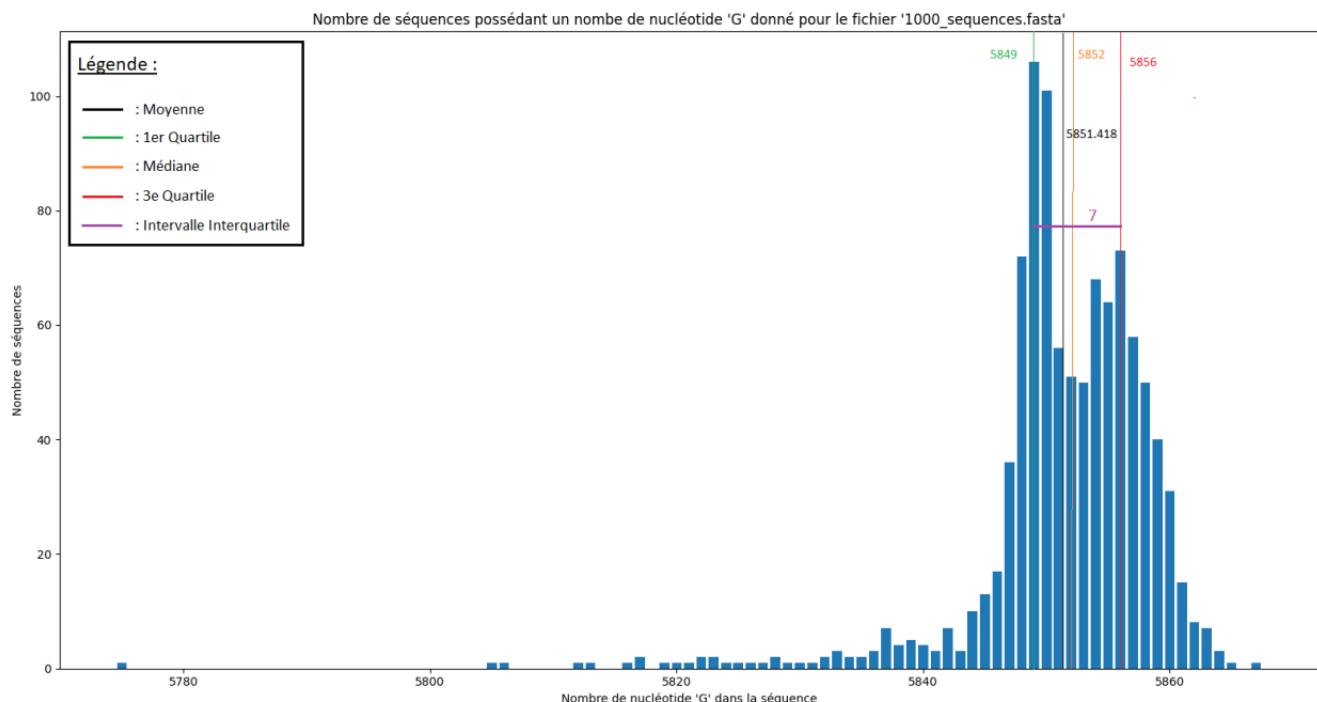


FIGURE 21 – La répartition des nombres de nucléotides G des séquences d'un échantillon de 1000

Diagramme circulaire de la proportion des nucléotides pour le fichier '1000\_sequences.fasta'

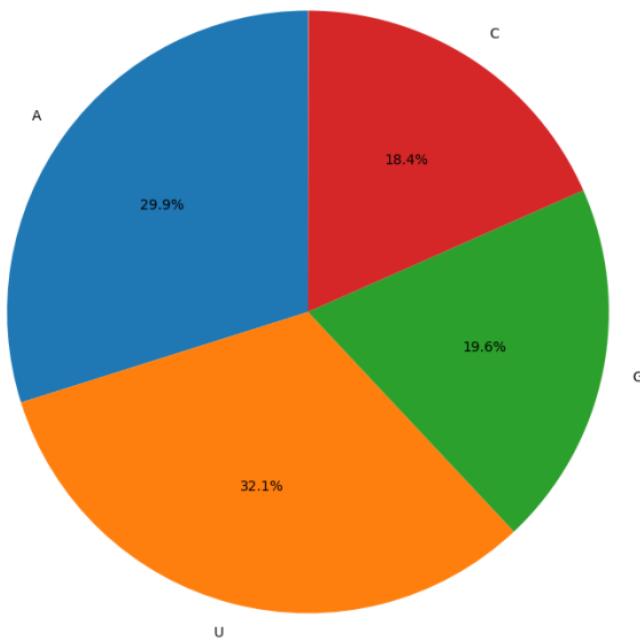


FIGURE 22 – Proportions des nucléotides dans un échantillon de 1000

La fonction `perform_all_stats_taille` prend en entrée une liste de séquences<sup>17</sup>, elle applique les fonctions statistiques en utilisant la fonction `call_stat_taille_genome`, et renvoie dans un dictionnaire les valeurs statistiques sur l'échantillon sur la taille du génome. Sur la figure 23, les résultats de deux applications de la fonction `perform_all_stats` sur deux échantillons de dix mille séquences de nucléotides de périodes différentes<sup>18</sup> sont mis en évidence.

```
In[10]: print(perform_all_stats_taille(fasta_to_genome("./genome/10000_sequences_janvier_aout_2020.fasta")))
Out[10]:
{'moy': 29803.0851, 'med': 29849.0, 'ecartt': 127.84136598921998, 'var': 16343.41485798969,
'quart1': 29794, 'quart3': 29868, 'int_quart': 74}

In[11]: perform_all_stats_taille(fasta_to_genome("./genome/10000_sequences.fasta"))
Out[11]:
{'moy': 29812.6683, 'med': 29813.0, 'ecartt': 43.368609328752584, 'var': 1880.8362751099658,
'quart1': 29796, 'quart3': 29831, 'int_quart': 35}
```

FIGURE 23 – Application de la fonction `perform_all_stats_taille` sur deux échantillons de 10000 séquences

De même que précédemment, les résultats des deux applications sur les deux échantillons diffèrent très peu. Nous avons en moyenne 29812 (avec plus ou moins 43 d'écart) nucléotides pour chaque séquence du génome. On retrouve les mêmes conclusions, la moyenne est assez proche de la médiane, l'écart interquartile n'est pas très grand, les valeurs sont plutôt symétriques et regroupées autour de la médiane. L'écart-type est plutôt petit par rapport à la taille des séquences, cependant pour l'échantillon de janvier à août, il est presque trois fois plus grand que celui d'octobre à novembre. Pareillement, cette différence peut s'expliquer par le fait que l'échantillon de janvier à août est d'une période plus large que celui d'octobre à novembre, le virus a pu muter beaucoup plus de fois, produisant des séquences qui s'écartent de plus en plus de la moyenne.

17. un échantillon

18. l'une sont des séquences de janvier à août 2020, et l'autre sont des séquences d'octobre à novembre 2020

Par ailleurs, une représentation graphique de l'application sur un échantillon de 1000 séquences a été réalisé, voir figure 24 et en comparaison avec les résultats précédents, les différences sont assez petites.

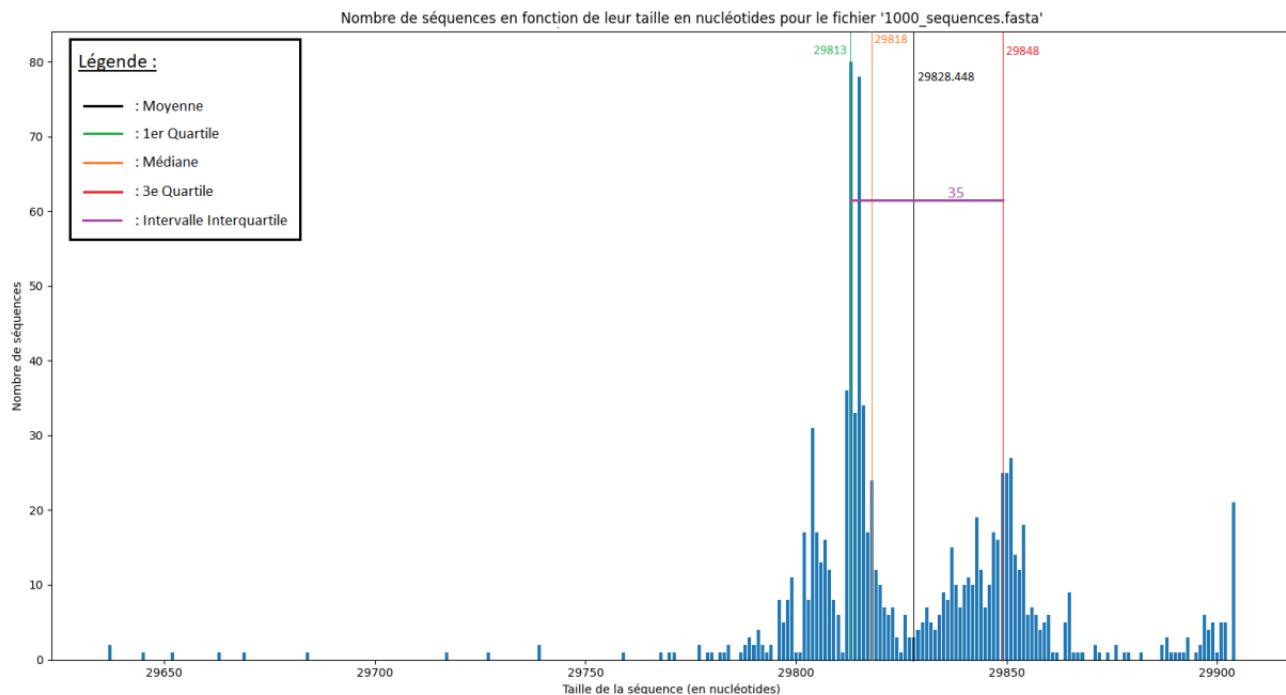


FIGURE 24 – La répartition de la taille des séquences de nucléotides dans un échantillon de 1000

## 2.2 La fonction codons

Le but de cette partie est d'écrire une fonction `codons` qui permet de générer les séquences d'acides aminés. La fonction doit prendre en paramètre une liste d'ARNm et renvoie la liste de la séquence d'acides aminés.

Pour l'implémentation de la fonction, les codons et leur acide aminé correspondant ont été ajoutés comme une constante globale dans le fichier `globals.py` sous forme d'un dictionnaire `CODONS_TO_AMINO_ACID`. La structure du dictionnaire a été choisie pour sa facilité d'appel et sa structure non-ordonnée.

Après de multiples documentations, un problème de traduction entre les codons START et STOP a été pris en compte. Donc une fonction `start_to_stop` a été implémentée pour sélectionner les séquences de l'ARNm sur lesquelles on va appliquer la fonction `codons`. Ces séquences sont retournées sous forme d'une liste de chaînes de caractères.

Toutefois, en appliquant la fonction sur les séquences du génome fournies par le site : *National Center for Biology Information (NCBI)*, un nouveau problème a été rencontré. Il y avait des codes IUPAC dans les fichiers FASTA autres que les quatres de base (A, G, C et U). Le problème c'est que ces codes, comme indiqué dans la table 3, page 12, peuvent avoir plusieurs traductions, comme ar exemple, le code N qui fait référence à n'importe quel nucléotide. On a donc pris en considération le code R (A ou G) dans la liste des codons STOP (UGA, UAA, UAG, UAR). Sachant que le codon UAR couvre les deux cas (UAA et UAG). Pour les autres codes plusieurs solutions ont été prises :

- L'implémentation d'une fonction `random` qui retourne aléatoirement le nucléotide : cette solution a été rejetée puisque dans des séquences il peut y avoir, par exemple, une succession de plusieurs

N. Donc, cela serait comme si on générerait toute une sous-séquence aléatoirement et les résultats seront erronées.

- L'étude de tous les cas, c'est-à-dire, à chaque code on fait une séquence pour chaque nucléotide possible. Cette solution aussi a été rejetée car cela serait très lourd en terme de mémoire et pour chaque séquence on aurait plus d'une centaine de séquences<sup>19</sup>.
- L'implémentation d'une fonction `try_AUGC` qui teste s'il y a un code autre que les nucléotides. Puis, on ignore tout codon contenant un code différent de (A, G, U, C et R). C'est la solution qui a été adaptée au final.

Finalement, la fonction `codons`, voir figure 25, prend en paramètre une chaîne de caractères ou une liste d'ARNm et renvoie une liste des séquences d'acides aminés qui correspondent à la traduction de chaque séquence entre les codons START et STOP tout en ignorant le problème des codes IUPAC.

```
codons('UUUUUCUUAUGUCUUCCUCAUCGUUUACUAAUAUGUGUUGCUGAUAG')
['MSSSSYY', 'MCC']

codons('UUUUUCUUAUGUCUUCCUCAUCGUUUACUAAUAGUGUUGCUGAUG')
['MSSSSYY']

codons(list('UUUUUCUUAUGUCUUCCUCAUCGUUUACUAAUAGUGUUGCUGAUG'))
['MSSSSYY']
```

FIGURE 25 – Application de la fonction `codons`

Mais, on a tout de même gardé la première version de `codons`, nommée `codons_v2`, voir figure 26, qui ne tient pas compte ni du problème de START et STOP ni celui des codes IUPAC autres que les basiques (A, G, C et U). Elle prend simplement en paramètre la liste d'ARNm et renvoie une chaîne des acides aminés.

```
codons_v2('UUUUUCUUAUGUCUUCCUCAUCGUUUACUAAUAUGUGUUGCUGAUAG')
'FFLCLPHRITNMCC**'

codons_v2('UUUUUCUUAUGUCUUCCUCAUCGUUUACUAAUAGUGUUGCUGAUG')
'FFLCLPHRITNSVAD'

codons_v2(list('UUUUUCUUAUGUCUUCCUCAUCGUUUACUAAUAGUGUUGCUGAUG'))
'FFLCLPHRITNSVAD'
```

FIGURE 26 – Application de la fonction `codons_v2`

Dû à l'ambiguïté du sujet, une troisième version de la fonction nommée `codons_v3` a été implémentée, voir figure 27, qui prend en paramètre l'ARNm et renvoie le même résultat que `codons` mais sous forme d'une chaîne de caractères. Cette version est utile pour les applications des analyses statistiques. Par conséquent, une fonction `codons_échantillon` a été implémentée, prenant en paramètre un échantillon du génome (plusieurs séquences d'ARNm) et renvoyant la liste des séquences d'acides aminés qui convient.

19. L'explosion combinatoire

```

codons_v3('UUUUUCUUAUGCUUCCUCAUCGUUUACUAAUAGUGUUGCUGAUAG')
'MSSSSYYMCC'

codons_v3('UUUUUCUUAUGCUUCCUCAUCGUUUACUAAUAGUGUUGCUGAUG')
'MSSSSYY'

codons_v3(list('UUUUUCUUAUGCUUCCUCAUCGUUUACUAAUAGUGUUGCUGAUG'))
'MSSSSYY'

```

FIGURE 27 – Application de la fonction codons\_v3

### 2.2.1 Étude de l'algorithme

L'algorithme de la fonction `codons` s'exécute en complexité linéaire. En effet, la fonction `start_to_stop` s'exécute en complexité linéaire. Pour chaque séquence entre START et STOP, de taille inférieure ou égale à la séquence d'ARNm initiale, on la parcourt avec un pas de trois.

### 2.2.2 Applications et interprétations

Nous avons appliqué les fonctions de l'analyse statistique descriptive sur les séquences d'acides aminés comme indiqué sur les figures 28 et 29 pour la fonction `perform_all_stats`. Un exemple graphique sur l'acide aminé L est illustré par la figure 30.

```

perform_all_stats(codons_echantillon(fasta_to_genome("./genome/1000_sequences_janvier_avril_2020.fasta")), AMINO_ACIDS)

{'moy': {'A': 112.514, 'R': 89.468, 'N': 80.968, 'D': 42.957, 'C': 96.977, 'Q': 135.921, 'E': 59.87, 'G': 83.836, 'H': 95.95,
'I': 119.986, 'L': 401.455, 'K': 121.53, 'M': 569.137, 'F': 152.023, 'P': 100.516, 'O': 0.0, 'S': 196.611, 'U': 0.0,
'T': 160.639, 'W': 53.932, 'Y': 106.029, 'V': 179.78, 'B': 0.0, 'Z': 0.0, 'X': 0.0, 'J': 0.0, '*': 0.0}, 

'med': {'A': 113.0, 'R': 90.0, 'N': 81.0, 'D': 43.0, 'C': 97.0, 'Q': 136.0, 'E': 60.0, 'G': 84.0, 'H': 96.0,
'I': 120.0, 'L': 402.0, 'K': 122.0, 'M': 569.0, 'F': 152.0, 'P': 101.0, 'O': 0.0, 'S': 197.0, 'U': 0.0,
'T': 161.0, 'W': 54.0, 'Y': 106.0, 'V': 180.0, 'B': 0.0, 'Z': 0.0, 'X': 0.0, 'J': 0.0, '*': 0.0}, 

'ecartt': {'A': 0.6751, 'R': 0.7648, 'N': 0.2549, 'D': 0.3242, 'C': 0.3556, 'Q': 0.5222, 'E': 0.6205, 'G': 0.5050, 'H': 0.3653,
'I': 0.4558, 'L': 1.2481, 'K': 0.7383, 'M': 1.0565, 'F': 0.5024, 'P': 0.6274, 'O': 0.0, 'S': 0.7859, 'U': 0.0,
'T': 0.8029, 'W': 0.3624, 'Y': 0.3408, 'V': 0.9162, 'B': 0.0, 'Z': 0.0, 'X': 0.0, 'J': 0.0, '*': 0.0}, 

'var': {'A': 0.4558, 'R': 0.5849, 'N': 0.0649, 'D': 0.1051, 'C': 0.1264, 'Q': 0.2727, 'E': 0.3850, 'G': 0.2551, 'H': 0.1334,
'I': 0.2078, 'L': 1.5579, 'K': 0.5450, 'M': 1.1162, 'F': 0.2524, 'P': 0.3937, 'O': 0.0, 'S': 0.6176, 'U': 0.0,
'T': 0.6446, 'W': 0.1313, 'Y': 0.1161, 'V': 0.8396, 'B': 0.0, 'Z': 0.0, 'X': 0.0, 'J': 0.0, '*': 0.0}, 

'quart1': {'A': 112, 'R': 89, 'N': 81, 'D': 43, 'C': 97, 'Q': 136, 'E': 60, 'G': 84, 'H': 96, 'I': 120, 'L': 401, 'K': 121, 'M': 569,
'F': 152, 'P': 100, 'O': 0, 'S': 196, 'U': 0, 'T': 160, 'W': 54, 'Y': 106, 'V': 180, 'B': 0, 'Z': 0, 'X': 0, 'J': 0, '*': 0}, 

'quart3': {'A': 113, 'R': 90, 'N': 81, 'D': 43, 'C': 97, 'Q': 136, 'E': 60, 'G': 84, 'H': 96, 'I': 120, 'L': 402, 'K': 122, 'M': 570,
'F': 152, 'P': 101, 'O': 0, 'S': 197, 'U': 0, 'T': 161, 'W': 54, 'Y': 106, 'V': 180, 'B': 0, 'Z': 0, 'X': 0, 'J': 0, '*': 0}, 

'int_quart': {'A': 1, 'R': 1, 'N': 0, 'D': 0, 'C': 0, 'Q': 0, 'E': 0, 'G': 0, 'H': 0, 'I': 0, 'L': 1, 'K': 1, 'M': 1, 'F': 0, 'P': 1,
'O': 0, 'S': 1, 'U': 0, 'T': 1, 'W': 0, 'Y': 0, 'V': 0, 'B': 0, 'Z': 0, 'X': 0, 'J': 0, '*': 0}}

```

FIGURE 28 – Échantillon de Janvier à Avril 2020

```

perform_all_stats(codons_echantillon(fasta_to_genome("./genome/1000_sequences.fasta")), AMINO_ACIDS)

{'moy': {'A': 112.796, 'R': 89.615, 'N': 80.999, 'D': 42.88, 'C': 96.974, 'Q': 135.736, 'E': 59.818, 'G': 83.801, 'H': 96.221,
'I': 120.092, 'L': 401.382, 'K': 121.65, 'M': 568.767, 'F': 151.905, 'P': 100.643, 'O': 0.0, 'S': 196.567, 'U': 0.0,
'T': 160.861, 'W': 53.938, 'Y': 105.612, 'V': 179.63, 'B': 0.0, 'Z': 0.0, 'X': 0.0, 'J': 0.0, '*': 0.0},
'med': {'A': 113.0, 'R': 90.0, 'N': 81.0, 'D': 43.0, 'C': 97.0, 'Q': 136.0, 'E': 60.0, 'G': 84.0, 'H': 96.0,
'I': 120.0, 'L': 402.0, 'K': 122.0, 'M': 569.0, 'F': 152.0, 'P': 101.0, 'O': 0.0, 'S': 197.0, 'U': 0.0,
'T': 161.0, 'W': 54.0, 'Y': 106.0, 'V': 180.0, 'B': 0.0, 'Z': 0.0, 'X': 0.0, 'J': 0.0, '*': 0.0},
'ecartt': {'A': 1.1074, 'R': 0.9299, 'N': 0.4826, 'D': 0.4995, 'C': 0.4328, 'Q': 0.7786, 'E': 0.6818, 'G': 0.6367, 'H': 0.8355,
'I': 0.7520, 'L': 1.6846, 'K': 0.7794, 'M': 1.1237, 'F': 0.7758, 'P': 0.6095, 'O': 0.0, 'S': 1.0007, 'U': 0.0,
'T': 0.7011, 'W': 0.3579, 'Y': 0.5562, 'V': 1.2865, 'B': 0.0, 'Z': 0.0, 'X': 0.0, 'J': 0.0, '*': 0.0},
'var': {'A': 1.2263, 'R': 0.8647, 'N': 0.2329, 'D': 0.2495, 'C': 0.1873, 'Q': 0.6063, 'E': 0.4648, 'G': 0.4053, 'H': 0.6981,
'I': 0.5655, 'L': 2.8380, 'K': 0.6074, 'M': 1.2627, 'F': 0.6019, 'P': 0.3715, 'O': 0.0, 'S': 1.0015, 'U': 0.0,
'T': 0.4916, 'W': 0.1281, 'Y': 0.3094, 'V': 1.6550, 'B': 0.0, 'Z': 0.0, 'X': 0.0, 'J': 0.0, '*': 0.0},
'quart1': {'A': 112, 'R': 89, 'N': 81, 'D': 43, 'C': 97, 'Q': 136, 'E': 60, 'G': 84, 'H': 96, 'I': 120, 'L': 401, 'K': 121, 'M': 569,
'F': 152, 'P': 100, 'O': 0, 'S': 196, 'U': 0, 'T': 161, 'W': 54, 'Y': 105, 'V': 180, 'B': 0, 'Z': 0, 'X': 0, 'J': 0, '*': 0},
'quart3': {'A': 113, 'R': 90, 'N': 81, 'D': 43, 'C': 97, 'Q': 136, 'E': 60, 'G': 84, 'H': 97, 'I': 120, 'L': 402, 'K': 122, 'M': 569,
'F': 152, 'P': 101, 'O': 0, 'S': 197, 'U': 0, 'T': 161, 'W': 54, 'Y': 106, 'V': 180, 'B': 0, 'Z': 0, 'X': 0, 'J': 0, '*': 0},
'int_quart': {'A': 1, 'R': 1, 'N': 0, 'D': 0, 'C': 0, 'Q': 0, 'E': 0, 'G': 0, 'H': 1, 'I': 0, 'L': 1, 'K': 1, 'M': 0, 'F': 0, 'P': 1,
'O': 0, 'S': 1, 'U': 0, 'T': 0, 'W': 0, 'Y': 1, 'V': 0, 'B': 0, 'Z': 0, 'X': 0, 'J': 0, '*': 0}}

```

FIGURE 29 – Échantillon d'Octobre à Novembre 2020

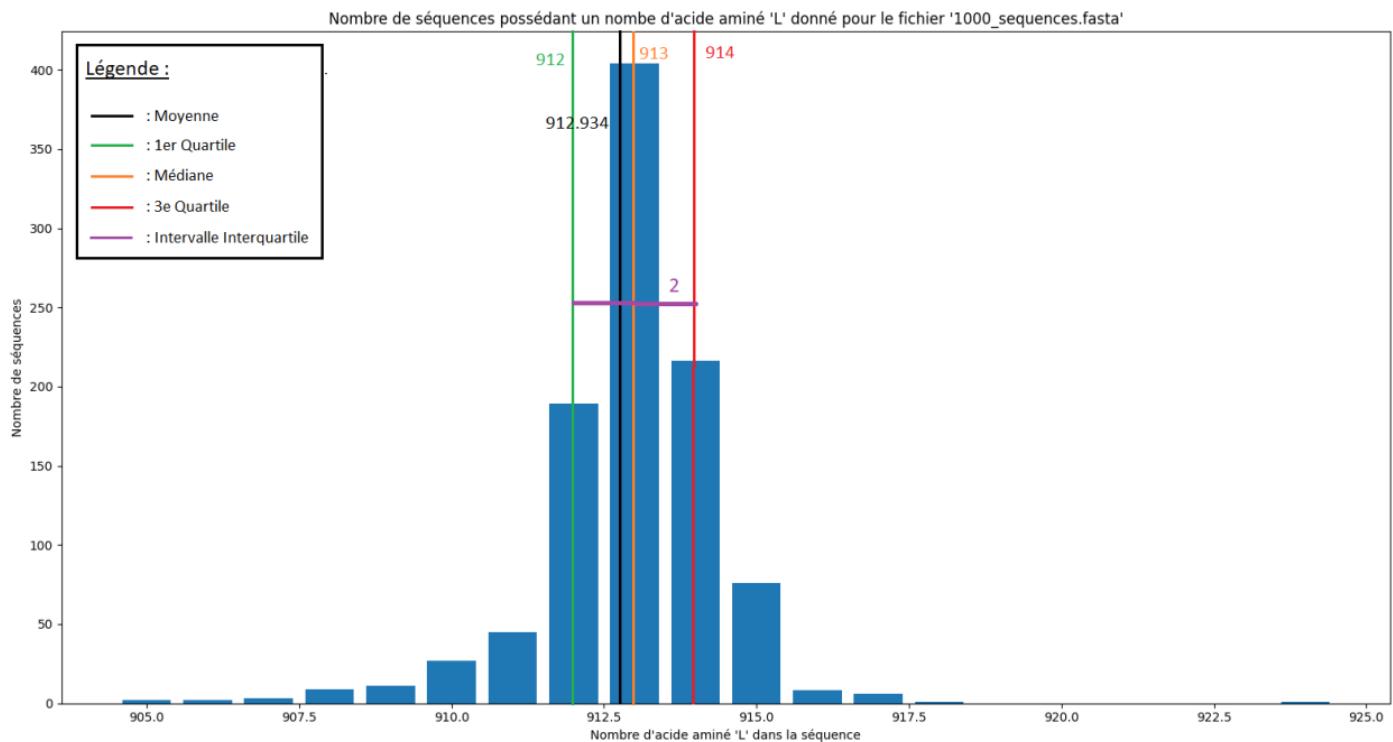


FIGURE 30 – Nombres d'acides aminés L des séquences d'un échantillon de 1000 séquences

On remarque que pour deux échantillons de périodes différentes, les résultats des analyses statistiques sont comparables.

Sur la figure 31, on trouve un exemple d'application de la fonction `perform_all_stats_proportion`.

Diagramme circulaire de la proportion des acides aminés pour le fichier '1000\_sequences.fasta'

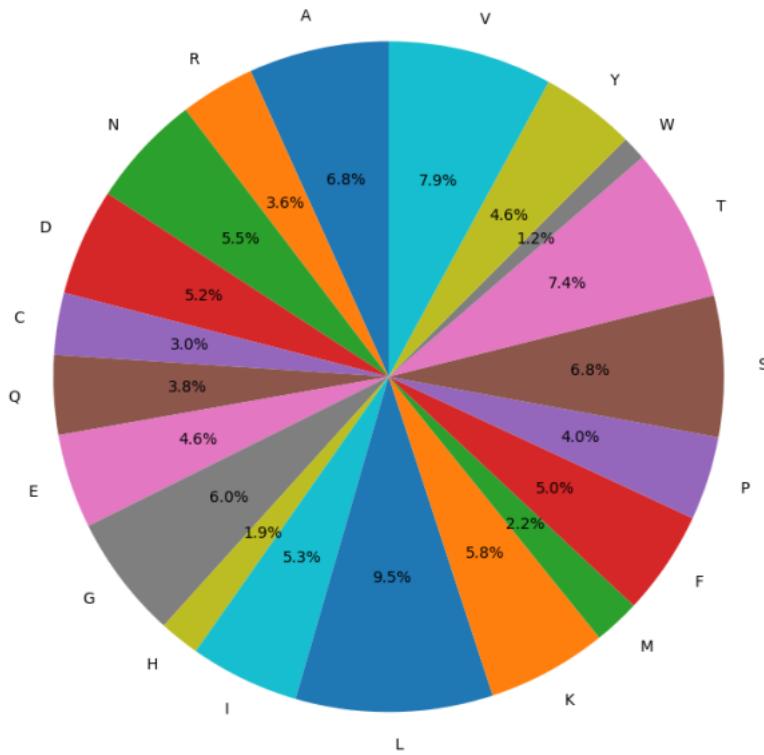


FIGURE 31 – Proportions des acides aminés dans un échantillon de 1000 séquences

Nous avons appliqué aussi la fonction `perform_all_stats_taille`. Sur les deux figures 32 et 33, les résultats de l'application de la fonction sur deux échantillons : l'un de mille séquences d'acides aminés et l'autre de dix milles séquences d'acides aminés.

```
perform_all_stats_taille(codons_echantillon(fasta_to_genome("./genome/1000_sequences.fasta")))
{'moy': 9612.965, 'med': 9617.0, 'ecartt': 11.420673141282043, 'var': 130.43177500000104,
 'quart1': 9615, 'quart3': 9617, 'int_quart': 2}
```

FIGURE 32 – Application de la fonction `perform_all_stats_taille` sur 1000 séquences d'acides aminés

```
perform_all_stats_taille(codons_echantillon(fasta_to_genome("./genome/10000_sequences.fasta")))
{'moy': 9598.714, 'med': 9617.0, 'ecartt': 251.38185893979113, 'var': 63192.839004025045,
 | 'quart1': 9616, 'quart3': 9617, 'int_quart': 1}
```

FIGURE 33 – Application de la fonction `perform_all_stats_taille` sur 10000 séquences d'acides aminés

La figure 34 illustre l'histogramme de nombre de séquence en fonction de leur taille en acides aminés.

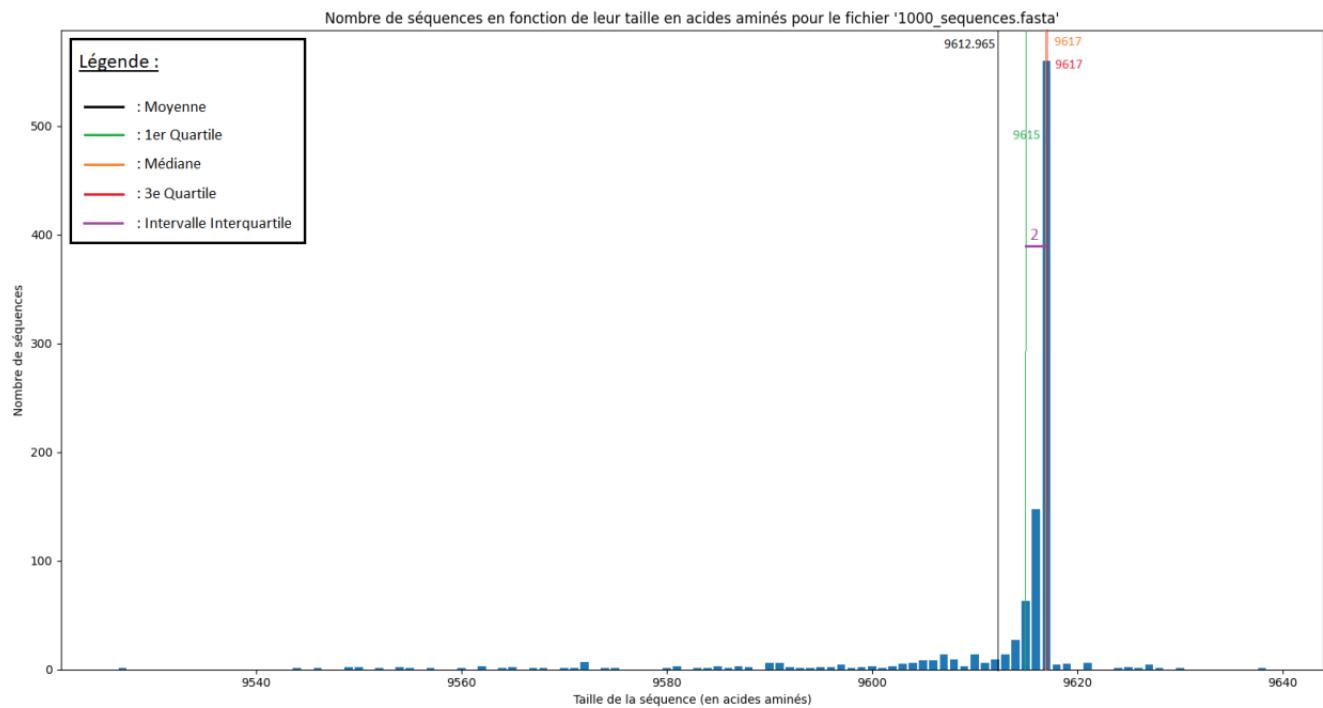


FIGURE 34 – Nombre de séquences en fonction de leur taille en acides aminés dans un échantillon de 1000

## 2.3 La distance de Levenshtein

Nous avons implémenté trois algorithmes pour le calcul de la distance de Levenstein : une version récursive classique `lev_rec`, une version en programmation dynamique récursive `lev_dp` et une version en programmation dynamique itérative `lev`, qui est notre version privilégiée. Toutes les trois prennent en paramètre deux chaînes de caractères, `seq1` et `seq2`, et renvoient un entier positif correspondant à la distance de Levenstein entre les deux chaînes de caractères saisies en entrée.

### 2.3.1 `lev_rec`

La fonction `lev_rec` est une version récursive du calcul de la distance de Levenstein qui ne fait pas appel à la programmation dynamique. L'algorithme est assez simple, on distingue quatre cas.

On a deux cas de base : si la longueur de `seq1` est égale à 0 ou si la longueur de `seq2` est égale à 0, alors on renvoie la longueur de `seq2` ou la longueur de `seq1` respectivement. C'est normal puisque si un des mots est vide, la distance de Levenstein entre les deux correspond à la longueur du second mot.

Troisième cas : si la première lettre de `seq1` est la même que la première lettre de `seq2`, alors on fait un appel récursif de `lev_rec` en prenant en paramètre `seq1` et `seq2` privées de leur première lettre. En effet, si ces deux premières lettres sont identiques, les enlever des deux mots ne change pas la distance de levenstein entre les deux.

Quatrième cas : si ces deux premières lettres ne sont pas identiques, on incrémente la distance de Levenshtein en ajoutant 1 au minimum des appels récursifs de `lev_rec` avec `seq1` privée de sa première lettre, `lev_rec` avec `seq2` privée de sa première lettre et `lev_rec` avec `seq1` et `seq2` privées de leur première lettre. Ceci correspond à prendre en compte le mismatch (+1) et à y ajouter la distance de Levenshtein minimale entre les deux chaînes de caractères dont l'une des deux ou les deux se voient privées de leur première lettre. Nous sommes obligés de prendre le minimum entre ces trois cas car ils permettent récursivement de couvrir toutes les possibilités.

Au final, la fonction renvoie bien la distance de Levenshtein entre les deux chaînes de caractères `seq1` et `seq2`. Cependant, l'obligation de multiplier les appels récursifs mène à une explosion combinatoire pour de très longues chaînes de caractères, ce qui peut mener à une erreur de "Stack overflow". En effet, dans le pire des cas où les chaînes de caractères sont complètement différentes, cet algorithme effectue trois appels récursifs et le fait en réduisant alternativement la taille de `seq1` et `seq2`, pour un total de  $m+n-1$  fois (car on s'arrête lorsqu'une des chaînes de caractères est vide). Ainsi, la complexité de cet algorithme est majorée en  $O(3^{n+m})$ , et c'est pourquoi cette solution n'est pas fiable pour l'étude de génomes, dont la taille est trop importante.

### 2.3.2 `lev_dp`

La fonction `lev_dp` est une version récursive du calcul de la distance de Levenshtein qui fait appel à la programmation dynamique.

Puisque l'on utilise la programmation dynamique, l'algorithme commence par créer une matrice de dimension  $(n+1) \times (m+1)$  comme décrit dans l'état de l'art. Ici, nous la remplissons de -1, ce qui permet à l'algorithme de savoir si une case a déjà été calculée ou pas.

Ainsi, on distingue cinq cas, dont trois cas de base :

- Premier cas (cas de base) : Si la dernière case de la matrice `dp[n+1][m+1]` n'est pas égale à -1, cela veut dire qu'elle a été calculée. Puisque cette case correspond à la distance de Levenshtein recherchée, on renvoie cette valeur.
- Deuxième cas (cas de base) : Si la longueur de `seq1` est nulle, cela veut dire que la distance de Levenshtein correspond à la longueur de `seq2`, donc on renvoie cette valeur.
- Troisième cas (cas de base) : Idem que pour le deuxième cas mais en inversant `seq1` et `seq2`.
- Quatrième cas : Si les premières lettres de `seq1` et `seq2` sont identiques, alors on attribue à `dp[n+1][m+1]` la valeur de l'appel récursif de la fonction auxiliaire `lambda_lev_rec` avec en paramètre les chaînes de caractères `seq1` et `seq2` privées de leur premières lettres. Au fil des appels récursifs, on tombera sur un des cas de base et ainsi `lambda_lev_rec` prendra bien la valeur de la distance de Levenshtein calculée. On retourne cette valeur.
- Cinquième cas : Si aucun des cas précédents n'est vérifié, alors c'est que l'on doit effectuer une addition, une suppression ou une substitution. On attribue donc à la dernière case de la matrice `dp[n+1][m+1]` la valeur  $1 + \text{la valeur minimale entre les appels récursifs de la fonction auxiliaire } \text{lambda\_lev\_rec} \text{ avec en paramètre } \text{seq1} \text{ et/ou } \text{seq2} \text{ privées de leur premières lettres}$  ainsi que la matrice `dp`, ce qui permet de couvrir tous les cas. Au fil des appels récursifs, on tombera sur un des cas de base et ainsi `lambda_lev_rec` prendra bien la valeur de la distance de Levenshtein calculée. On retourne cette valeur.

Ainsi, on calcule récursivement la distance de Levenshtein mais en utilisant une matrice "dp" qui fait office de "sauvegarde" des données déjà calculées. Ainsi, dans le pire cas on n'effectue que  $(n+1) \times (m+1) = n \times m + n + m + 1$  opérations au maximum, ce qui est majoré en  $O(n \times m)$ . Ainsi, cette fonction est théoriquement largement plus efficace que `lev_rec` pour des chaînes de caractères plus longues. Cependant, il existe une solution encore meilleure, présentée ci-après.

### 2.3.3 lev

La fonction `lev` est une version itérative du calcul de la distance de Levenstein qui fait aussi appel à la programmation dynamique. Son fonctionnement est le plus simple parmi les trois versions proposées, et aussi la plus efficace. Notez que le déroulement ne suit pas exactement celui énoncé dans l'état de l'art. Ici, l'étape d'initialisation est directement incorporée dans l'étape de remplissage, cela a quelques conséquences sur les indices de la boucle.

La fonction commence par créer une matrice nommée "dp" de dimension  $(m + 1) \times (n + 1)$  remplie de -1. Ici, la valeur -1 n'a aucune importance. On rentre ensuite dans la double boucle parcourant chaque case de la matrice "dp" et on distingue quatre cas :

- Premier et deuxième cas : Si  $i = 0$ , alors `dp[i][j]` prend la valeur  $j$  et sinon si  $j = 0$  alors `dp[i][j]` prend la valeur  $i$  (ce qui correspond à l'étape d'initialisation).
- Troisième cas : Si les deux cas précédents ne sont pas vérifiés et si `seq1[i - 1] = seq2[j - 1]`, alors `dp[i][j]` prend la valeur `dp[i-1][j-1]`. Ce cas correspond à un match mais comparé à l'état de l'art, on compare les lettres `seq1[i-1]` avec `seq2[j-1]` au lieu de comparer `seq1[i]` avec `seq2[j]`. En effet, la matrice "dp" doit forcément être de dimension  $(m + 1) \times (n + 1)$  pour inclure le "gap" devant les mots, et donc  $i$  et  $j$  varient de 1 à  $n + 1$  et  $m + 1$ . Or `seq1` et `seq2` ne sont que de longueur  $n$  et  $m$  puisque nous n'avons pas ajouté artificiellement le gap devant les chaînes de caractères. Nous sommes donc obligés de soustraire 1 aux indices  $i$  et  $j$  dans `seq1[i-1]` et `seq2[j-1]`. En somme, cela revient à mettre les gaps à la fin des mots.
- Quatrième cas : Si aucun des cas précédents ne soient vérifiés, alors `dp[i][j]` prend la valeur  $1 + \min(\text{cas à gauche}, \text{cas au dessus}, \text{cas diagonale en haut})$  (`dp[i-1][j]`, `dp[i][j-1]`, `dp[i-1][j-1]`)

Au final, on renvoie `dp[n][m]` qui est notre distance de Levenstein. Le fait d'implémenter une matrice "dp" de dimension  $(m + 1) \times (n + 1)$  permet de réduire la complexité en  $O(m \times n)$ . En effet, quelque soient les longueurs des chaînes de caractères, l'algorithme effectue  $(m + 1) \times (n + 1) = m \times n + m + n + 1$  opérations, ce qui est majoré en  $O(m \times n)$ . De plus, d'après les graphiques de performance, cette version est plus rapide que la version récursive en programmation dynamique `lev_dp`, bien qu'elle soit elle aussi majorée en  $O(m \times n)$ . Nous pensons que cela est du à un coût supplémentaire par rapport à la gestion de la pile due aux appels récursifs successifs de la fonction. C'est pourquoi `lev` est notre fonction la plus efficace pour le calcul de la distance de Levanstein.

### 2.3.4 Applications et interprétation

La fonction `meme_taille` prend en argument deux listes de chaînes de caractères de tailles différentes et qui retourne deux listes de la même taille en ajoutant des chaînes vides. La fonction `affiche_res` a été implémentée pour l'affichage du résultat. Elle prend en argument les deux séquences d'acides aminés<sup>20</sup>.

Différentes applications ont été faites en comparant à chaque fois les séquences de codons selon les critères de la période et du lieu :

- Différentes périodes et différents lieu :

- La comparaison est entre une première séquence prise de la Chine, la séquence "originelle" de Wuhan en janvier 2020, et une deuxième séquence prise des Etats Unis en décembre 2020. Comme indiqué sur la figure 35, on remarque que le début des séquences est identiques mais à partir d'un endroit il y a beaucoup de changement (les mutations) au milieu et à la fin des séquences.

20. obtenues par l'application de la fonction `codons` sur deux ARNm de la base NCBI.

FIGURE 35 – Application de la fonction lev sur les séquences des Etats-Unis en décembre 2020 et de Chine en janvier 2020

- Une comparaison entre une première séquence prise du Brésil en Mars 2020 et une deuxième séquence prise des États-Unis en décembre 2020 est montrée sur la figure 36. Les deux séquences sont presque identiques, il n'y a que deux ou trois changements.

FIGURE 36 – Application de la fonction `lev` sur les séquences des Etats-Unis en décembre 2020 et du Brésil en Mars 2020

- Même période :
    - Une comparaison entre deux séquences proches en période, de Belgique et d'Allemagne en décembre 2020<sup>21</sup>. Les résultats sont représentés sur la figure 37, les séquences diffèrent très peu.
    - Une comparaison entre deux séquences de même période, mais de lieux éloignés, des Etats-Unis et de la Tunisie en décembre 2020, a été fait. Elles diffèrent peu également.
  - Même lieu :
    - Une comparaison entre deux séquences de même lieu (la Chine) et de périodes différentes. L'une en janvier 2020 et l'autre en mars 2020 montre que les séquences sont identiques avec juste un changement dans la dernière partie (voir figure 38).
    - Une comparaison entre deux séquences de même lieu (les Etats-Unis Californie) et de périodes différentes décrit que les séquences sont presque identiques au début avec des changements sur les dernières séquences d'acides aminés.
  - Même période et même lieu :
    - Deux séquences (choisies au hasard) de même lieu (la Chine) et de mêmes périodes (mars 2020) sont presque identiques.

21. et proche en lieu aussi

FIGURE 37 – Application de la fonction `lev` sur les séquences de Belgique et d'Allemagne en décembre 2020

FIGURE 38 – Application de la fonction `lev` sur deux séquences différentes de la Chine en mars 2020

- Une comparaison entre deux séquences de même lieu (les Etats-Unis Californie) et de mêmes périodes (décembre 2020) montre qu'elles sont identiques sauf à la fin l'une est plus longue que l'autre.

En conclusion, on peut remarquer que dans le plus part du temps la mutation s'effectue à la fin de la séquence du génome soit par changement des séquences d'acides aminés, soit par l'ajout de ces dernières (justifié par différence de taille des séquences).

### 2.3.5 Pistes d'amélioration possibles

Comme mentionné dans l'état de l'art, il serait possible d'améliorer les fonctions en y ajoutant un moyen de personnaliser les coûts d'addition, de suppression et de substitution d'un caractère. De plus, on pourrait effectuer le calcul en ne gardant que la ligne précédente plutôt que de sauvegarder tout le tableau, ce qui permettrait de gagner un peu d'espace mémoire.

## 2.4 L'algorithme de Needleman-Wunsch

L'algorithme de Needleman-Wunsch doit avoir deux versions de fonctionnement (l'une qui prend la matrice de similarité<sup>22</sup> et l'autre qui prend juste le coût du *match*, *mismatch* et le *gap*). Afin de profiter des similitudes entre ces deux versions et dans le but d'éviter les répétitions des codes, les différents arguments ont été regroupés, aboutissant à la fin en une seule fonction polymorphe<sup>23</sup>. Pour une telle fonction, il est nécessaire de faire des vérifications supplémentaire qui s'assurent de la correcte utilisation de la fonction et qui garantissent l'exclusivité mutuelle de ces différents arguments.

La structure des arguments est identique pour `needleman` et `needleman_all`. Les deux premiers arguments sont respectivement la première et la deuxième séquence. Ensuite, nous passons soit une

22. ou matrice de coût

23. Le comportement de la fonction change légèrement en fonction des paramètres fournis, ce qui est similaire au concept de surcharge de fonction

table de coûts, soit une matrice de coûts et une clé. La table `cost_table`<sup>24</sup> est une liste qui contient les coûts de *match*, *mismatch* et le *gap* (dans cet ordre). L'ensemble composé de la matrice de coûts (`cost_matrix`) et de la clé (`key`) est mutuellement exclusive avec la `cost_table` (soit l'un, soit l'autre). La manière dont la `cost_matrix` est lue et interprétée dépend fortement de la clé et de sa valeur. Fondamentalement, la `cost_matrix` est un tableau python avec une longueur égale à la longueur de la clé au carré en ajoutant un (le dernier élément de la table `cost_matrix` qui est en fait le score de l'écart, le *gap*). Le premier élément de la table correspond au coût entre le premier caractère de la clé et lui-même, le deuxième élément est le coût entre le premier et le deuxième caractère de la clé, et ainsi de suite. Par exemple, nous avons la table et la clé suivante :

`cost_mat = [ -1, -2, -3, -4, -5, -6, -7, -8, -9, -10 ]` et `key = "ABC"`

Alors, la matrice de coûts (de similarité) sera dans la table 4 suivante :

		Letter 2		
		A	B	C
Letter 1	A	-1	-2	-3
	B	-4	-5	-6
	C	-7	-8	-9

TABLE 4 – Un exemple de matrice de similarité avec un `gap=-10`

Pourquoi cette disposition, pourquoi ne pas se contenter d'une table python multidimensionnelle ? Ce que vous pourriez demander. Plusieurs arguments peuvent être avancés pour justifier cette conception. Premièrement, la simplicité et la facilité d'utilisation. Deuxièmement, les performances. En effet, les tableaux linéaires sont plus conviviaux pour le cache<sup>25</sup> que les tableaux multidimensionnels et donc plus rapides d'accès, et il n'y a pas de perte de temps pour la récupération d'informations des caches de niveau supérieur, voire même de la mémoire principale.

Pour finir sur cela, il est important de mentionner que `needleman` a un argument supplémentaire nommé `verbose` qui est défini par défaut sur `False`. Ceci est utilisé pour renvoyer, en plus de l'alignement global, la matrice de score et le chemin emprunté pour construire l'alignement. Ils sont ensuite utilisés plus tard pour montrer une exécution animée de l'algorithme Needleman-Wunsch et la construction de l'alignement global pour répondre à la question 8 (voir `app/needleman_seq.py`).

#### 2.4.1 Implémentation de `needleman`

La fonction `needleman` renvoie un seul alignement avec son score. Pour faciliter l'accès aux coûts, une fonction lambda<sup>26</sup> appelée `get_cost(letter1, letter2)` a été définie. Puisque l'algorithme Needleman-Wunsch se compose de 3 étapes majeures qui sont l'initialisation, le remplissage et le trace back<sup>27</sup>. Le code a également été divisé en plusieurs morceaux pour effectuer ces étapes.

L'étape d'initialisation consiste essentiellement à définir la ligne la plus haute et la colonne la plus à gauche de la matrice de score initialement mise à zéro (car zéro est l'élément neutre de l'addition).

```
1 # Init step:
2 len_seq1, len_seq2 = len(seq1), len(seq2)
```

24. table de coûts

25. La mémoire cache du processeur

26. Les fonctions anonymes sont appelées lambda, le terme vient de Lambda-Calculs inventé par Alonzo Church en 1930.

27. le traçage

```

3 alignement_mat = [[0 for _ in range(len_seq1+1)] for _ in range(len_seq2+1)]
4
5 for i in range(1, len_seq1+1):
6     alignement_mat[0][i] = alignement_mat[0][i-1] + gap
7
8 for j in range(1, len_seq2+1):
9     alignement_mat[j][0] = alignement_mat[j-1][0] + gap

```

Listing 1 – Phase d’initialisation

L’étape de remplissage est également triviale car nous parcourons les cellules restantes de la matrice et les remplissons une par une. (Voir 1.4)

```

1 # Filling:
2 for j in range(1, len_seq2+1):
3     for i in range(1, len_seq1+1):
4         left_val = alignement_mat[j][i-1] + gap
5         up_val = alignement_mat[j-1][i] + gap
6         diag_val = alignement_mat[j-1][i-1] + get_cost(seq1[i-1], seq2[j-1])
7         alignement_mat[j][i] = max(left_val, up_val, diag_val)

```

Listing 2 – Phase de remplissage

Le traçage qui est la dernière étape est légèrement plus compliqué que les étapes précédentes. Nous définissons d’abord notre position actuelle sur la dernière cellule de notre matrice précédente. Ensuite, nous le mettons constamment à jour jusqu’à ce que nous atteignons la première cellule. Il est clair que cela peut être fait en utilisant une boucle `while` qui vérifie si nous avons atteint la cellule en haut à gauche. A chaque itération, nous mettons à jour la position actuelle en fonction de la situation dans laquelle nous nous trouvons actuellement. Il y a quatre cas majeurs que nous pouvons rencontrer lors de l’exécution de cette boucle :

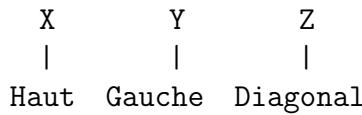
1. Le premier cas est le suivant : si nous atteignons la rangée supérieure de la matrice, nous continuons à prendre à gauche jusqu’à ce que nous atteignons la première cellule, en insérant un espace dans la deuxième séquence.
2. Le deuxième cas est similaire au premier : si nous atteignons la colonne la plus à gauche de la matrice, nous continuons à monter et à insérer un espace dans la première séquence jusqu’à ce que nous atteignons la première cellule.
3. Le troisième cas est le suivant : si la somme de la valeur voisine de la cellule diagonale et le coût entre deux caractères actuels est égale à la valeur de notre cellule actuelle. Alors, nous définissons notre position actuelle sur cette cellule diagonale et insérons les deux caractères dans leurs séquences respectives.
4. Le quatrième cas est celui où rien de ce qui précède ne se produit. Nous devrons, alors regarder les valeurs de la cellule voisine (en haut et à gauche) de notre cellule actuelle et ajouter la valeur de l’écart, si nous obtenons la valeur détenue par notre cellule actuelle, nous mettons à jour vers cette cellule voisine spécifique. Si nous allons dans la cellule de gauche, nous insérons un espace dans la deuxième séquence, de même si nous montons, nous insérons un espace dans la première séquence.

Suivant cette logique, la boucle `while` ne peut pas être infinie et se terminera forcément dans un temps fini. De plus, nous pouvons également remarquer que l’algorithme a une complexité théorique de  $\Theta(n.m)$ . Étant donné que l’étape de remplissage se compose de deux boucles `for` imbriquées, et que la trace back sera dans le pire des cas  $\Theta(n + m)$ .

### 2.4.2 Implémentation de needleman\_all

Alors que `needleman` ne renvoie qu'un seul alignement, `needleman_all` renverra tous les alignements (même ceux redondants<sup>28</sup>). La mise en œuvre des deux premières étapes de l'algorithme (initialisation et remplissage) est plus ou moins la même que ce qui a été mentionné précédemment. À l'exception du fait que nous avons maintenant défini ce que nous appelons une matrice de direction qui sera utilisée plus tard pour rendre le trace back un peu plus facile. La matrice de direction dans notre implémentation est une matrice qui a les mêmes dimensions que la matrice de score (`alignement_mat` dans le code) mais a un contenu différent. Au lieu de coder les directions à l'aide des chaînes (par exemple : `GAUCHE`, `HAUT` et `DIAGONAL`) ou des lettres (`G`, `H`, `D`) ou même des entiers (`1` : `Gauche`, `2` : `Haut`, `3` : `Diagonal`). Nous avons choisi de les coder en binaire, car c'était la solution ingénieuse dans ce cas. Puisque les directions peuvent être une combinaison de `GAUCHE`, `HAUT` et `DIAGONAL`, il y a au total  $C_3^1 + C_3^2 + C_3^3 = \text{card}(P(\{G, H, D\}) \setminus \emptyset) = 7$  possibilités. Pour calculer le nombre de bits dont nous avons besoin pour coder ces combinaisons d'états, nous pouvons prendre le  $\log_2(7)$  arrondi au plus grand entier le plus proche. Par conséquent :  $NbBits = \lceil \log_2(7) \rceil = \lceil 2.8 \rceil = 3$ . Ou sans utiliser l'équation mentionnée ci-dessus, nous pouvons remarquer que  $\text{card}(P(\{G, H, D\})) = 2^3 = 8$  on constate rapidement qu'il faut 3 bits et qu'un état est extra qui ne sera pas codé<sup>29</sup>.

La disposition des bits sera comme ceci :



Le premier bit indique si nous devons ou non aller en diagonale. Le deuxième indique si nous devons aller à gauche ou non. Et le troisième sert à monter en haut.

Cette solution est beaucoup plus efficace que celles mentionnées précédemment (au début de ce paragraphe), car celles-ci nécessiteront une allocation de mémoire dynamique puisqu'elles sont stockées dans des tables et rendront le code plus long et compliqué à lire. Lors de l'étape d'initialisation et de remplissage, nous remplissons également cette matrice de direction en utilisant les `shift`<sup>30</sup> et les `or` binaires. Par exemple, la première ligne sera remplie avec `010` (qui est `2` en décimal), de même la colonne la plus à gauche sera remplie avec `100` (`4` en décimal). Les autres cellules sont remplies en fonction de la valeur des cellules voisines et des coûts de fonctionnement.

```

1 if diag_val == alignement_mat[j][i]:
2     mat_dir[j][i] |= 1
3 if left_val == alignement_mat[j][i]:
4     mat_dir[j][i] |= (1 << 1)
5 if up_val == alignement_mat[j][i]:
6     mat_dir[j][i] |= (1 << 2)

```

Listing 3 – Définition des bits de direction des cellules 'intérieures' de la matrice. Ce morceau de code est inséré à l'intérieur d'une boucle `for` imbriqué similaire à celle mentionnée dans la section précédente

Passons à la troisième et dernière étape, la trace back, qui sera plus compliquée que la variante utilisée dans `needleman` car maintenant nous sommes obligés de parcourir tous les chemins. Si nous réfléchissons à ce problème, nous constaterons qu'il est plus ou moins équivalent à un parcours de graphe. Et nous savons qu'il existe deux types de parcours de graphe, la recherche en largeur (BFS<sup>31</sup>)

28. Par redondant, nous ne voulons pas dire des doublons mais des alignements qui sont un équivalents comme (`GA-AT`, `G-GAT`) et (`G-AAT`, `GG-AT`) pour un score de gap nul

29. l'ensemble vide ou en d'autres termes l'absence de directions

30. décalage binaire vers la gauche

31. Breadth First Search

et la recherche en profondeur (DFS<sup>32</sup>). Nous avons opté pour ce dernier. Tout ce dont nous avons besoin maintenant est une structure de données FIFO<sup>33</sup> pour stocker les coordonnées là où il y a une bifurcation et une copie de la chaîne construite jusqu'à ce point, et nous pouvons commencer le parcours de graphe. Nous commençons par pousser les premières coordonnées vers la FIFO. Et tant que le FIFO n'est pas vide, nous prenons la première paire de coordonnées avec sa séquence construite (initialement vide) du FIFO. Et nous continuons la construction de l'alignement en utilisant la matrice de directions que nous avons construite dans les deux premières étapes de l'algorithme. Si nous rencontrons plus d'une direction possible, nous sauvegardons une copie de la séquence construite avec les coordonnées suivantes, en prenant cette direction, dans la FIFO. Ensuite, nous mettons à jour les coordonnées actuelles jusqu'à atteindre la cellule en haut à gauche où nous ajoutons la chaîne construite (la variable `path` dans le code) à la variable `output`<sup>34</sup>. Et puis à la fin, lorsque nous atteignons la première cellule, nous sortons le premier élément de la FIFO et faisons le même type de traitement sur les coordonnées suivantes dans la FIFO jusqu'à ce qu'elle soit vide. L'algorithme se termine dans un temps fini car il y a un nombre limité de chemins entre la cellule en bas à droite et la cellule en haut à gauche.

Le calcul de la complexité de l'algorithme sera plus compliqué que la version `needleman`. Nous pouvons commencer par dire que les deux premières étapes de l'algorithme sont  $\Theta(n.m)$ . Mais la complexité du retraçage est différente, elle est liée au nombre de chemins différents pouvant être empruntés de la cellule inférieure droite à la cellule supérieure gauche, en tenant compte du fait que les seuls mouvements autorisés sont vers le haut, la gauche ou la diagonale. Il s'avère que le nombre de chemins dans cette situation s'appelle le nombre de Delannoy<sup>35</sup>. Ce nombre indique le nombre de chemins possibles dans une grille rectangulaire de longueur  $n$  et largeur  $m$ , en se déplaçant vers la cellule supérieure, gauche ou diagonale. Ce nombre est défini comme suit :

$$D(n, m) = \begin{cases} 1 & \text{si } m = 0 \text{ ou } n = 0 \\ D(m - 1, n - 1) + D(m - 1, n) + D(m, n - 1) & \text{sinon} \end{cases}$$

Donc la complexité de l'algorithme est  $O(D(n, m))$  puisque  $D(n, m)$  est beaucoup plus grand que  $n.m$  qui le coût des deux étapes précédentes de l'algorithme. Cependant nous savons que :  $D(n, m) = O(e^{n+m})$  [24]. Nous pouvons donc affirmer avec certitude que la complexité globale de l'algorithme est exponentielle au pire des cas de l'ordre de  $O(e^{n+m})$ . Au mieux et en moyenne, la complexité sera  $\Omega(n.m)$  quand il y a pas beaucoup de chemins possibles.

#### 2.4.3 Exécution animée de la fonction `needleman`

Pour répondre à la question 8, nous devons fournir une exécution animée de l'algorithme de Needleman-Wunsch. Comme nous n'étions pas autorisés à utiliser des bibliothèques externes en dehors de celles mentionnées dans le sujet, nous avons décidé de tout animer sur le terminal. Pour cela, plusieurs fonctions d'assistance doivent être définies avant. Certaines de ces fonctions sont :

- `clear` : cette fonction dépend de la plate-forme<sup>36</sup>. Son travail est d'effacer la console. Nous utilisons cette fonction juste avant de dessiner.
- `create_aligner_str` : cette fonction prend deux alignements de séquences et les décore en dessinant les *match* et les *mismatch* en lignes verticales.

32. Depth First Search

33. First In First Out

34. qui est un tableau de tous les alignements possibles et de leur score

35. Officier de l'armée française et mathématicien amateur [23]

36. l'OS, le système d'exploitation

- **color** : est une fonction utilisée pour ajouter une couleur à une chaîne spécifique. l'énumération<sup>37</sup> **COLORS** définit plusieurs caractères spéciaux de couleur utilisés dans le terminal.
- **spacer** : étant donné un certain nombre d'espaces et une chaîne, le **spacer** centre la chaîne entre les espaces dans le terminal.
- **print\_footer** : est une fonction d'aide qui est utilisée plus tard pour imprimer le développement et la construction de l'alignement.
- **print\_nw\_table** : affiche le développement de la matrice de score mettant en évidence le chemin emprunté avec une couleur. Ensuite, elle affiche, en bas de la matrice, l'évolution de l'alignement de séquence.
- **nw\_verbose** : est la fonction finale qui doit être utilisée pour rassembler toutes les pièces pour montrer l'exécution animée de l'algorithme. La fonction appelle d'abord **needleman** avec le paramètre **verbose** défini sur **True**. Ce qui renvoie un alignement, une matrice de score et le chemin emprunté pour construire l'alignement. Pour chaque coordonnée du chemin, la console sera effacée, pour créer l'illusion d'une animation, et la table sera imprimée à l'aide de **print\_nw\_table**. Le programme attend quelques secondes, cela est paramétré par un **timer**, et par la fonction **sleep**. Temporiser l'exécution du programme est important, car cela permettra à l'utilisateur de voir la progression du traçage, sinon la fonction s'exécutera instantanément.

#### 2.4.4 Application de Needleman-Wunsch

Dans un article du 25 décembre 2020 issu du site web berthub.eu [25], il est question d'une présentation du vaccin contre le SARS-CoV-2 mis au point par le laboratoire Pfizer/BioNTech. On y trouve notamment un lien menant à la séquence d'ARNm utilisé dans le vaccin, correspondant au gène codant pour la protéine "spike" du virus.

Un vaccin consiste à injecter chez un individu une petite partie du pathogène responsable de la maladie à traiter. En faisant cela, le corps développe une réponse immunitaire face à cette partie du pathogène et crée des anticorps menant à leur destruction. Ainsi, lors d'une vraie infection par le pathogène, le système immunitaire possède déjà les anticorps nécessaires pour combattre le pathogène, c'est ce que l'on appelle l'immunité face à ce pathogène.

Ainsi, cet ARNm présent dans le vaccin de Pfizer/BioNTech doit correspondre à une partie du génome du virus du SARS-CoV-2. Nous nous proposons donc de retrouver l'emplacement exact de cet ARNm dans le génome du coronavirus SARS-CoV-2 en utilisant l'algorithme de Needleman-Wunsch.

Voici ci-dessous un aperçu de l'alignement effectué par notre algorithme **needleman** avec en paramètres un génome du SARS-CoV-2, l'ARNm présent dans le vaccin et une table de coûts de +5 par match, -5 par mismatch et -1 par gap :

D'après cet alignement, la région du génome correspondant à l'ARNm du vaccin se situe entre les nucléotides n°11330 jusqu'à la fin (le génome possède au total 29903 nucléotides). Cependant, on constate un nombre important de gaps séparant les nucléotides du vaccin. Ceci est assez peu précis pour localiser précisément l'emplacement gène "spike" dans le génome du virus.

37. En réalité, cela est déclaré comme une classe en python car il ne prend pas en charge les énumérations

FIGURE 39 – Extrait de l'alignement du génome du SARS-CoV-2 avec l'ARNm du vaccin de Pfizer/BioNTech. (Match = 5, Mismatch = -5, gap = -1)

Voici ci-dessous un autre alignement, mais cette fois-ci en utilisant un aligneur de séquences en ligne [26] :

FIGURE 40 – Extraits de l’alignement du génome du SARS-CoV-2 avec l’ARNm du vaccin de Pfizer/BioNTech. (EMBOSS)

Ici, on constate tout de suite une quasi absence de gaps entre les nucléotides de la séquence d'ARNm du vaccin. Ceci induit une localisation beaucoup plus exacte du gène "spike" dans le génome du SARS-CoV-2.

D'après cet alignement, le gène semble commencer au 21506e nucléotide du génome, et semble se terminer vers le 25739e nucléotide (en ne comptant pas la queue poly-A de la séquence d'ARNm).

Voici un document issu du National Center for Biotechnological Information (NCBI) montrant précisément la localisation du gène "spike" pour le SARS-CoV-2 [27] :

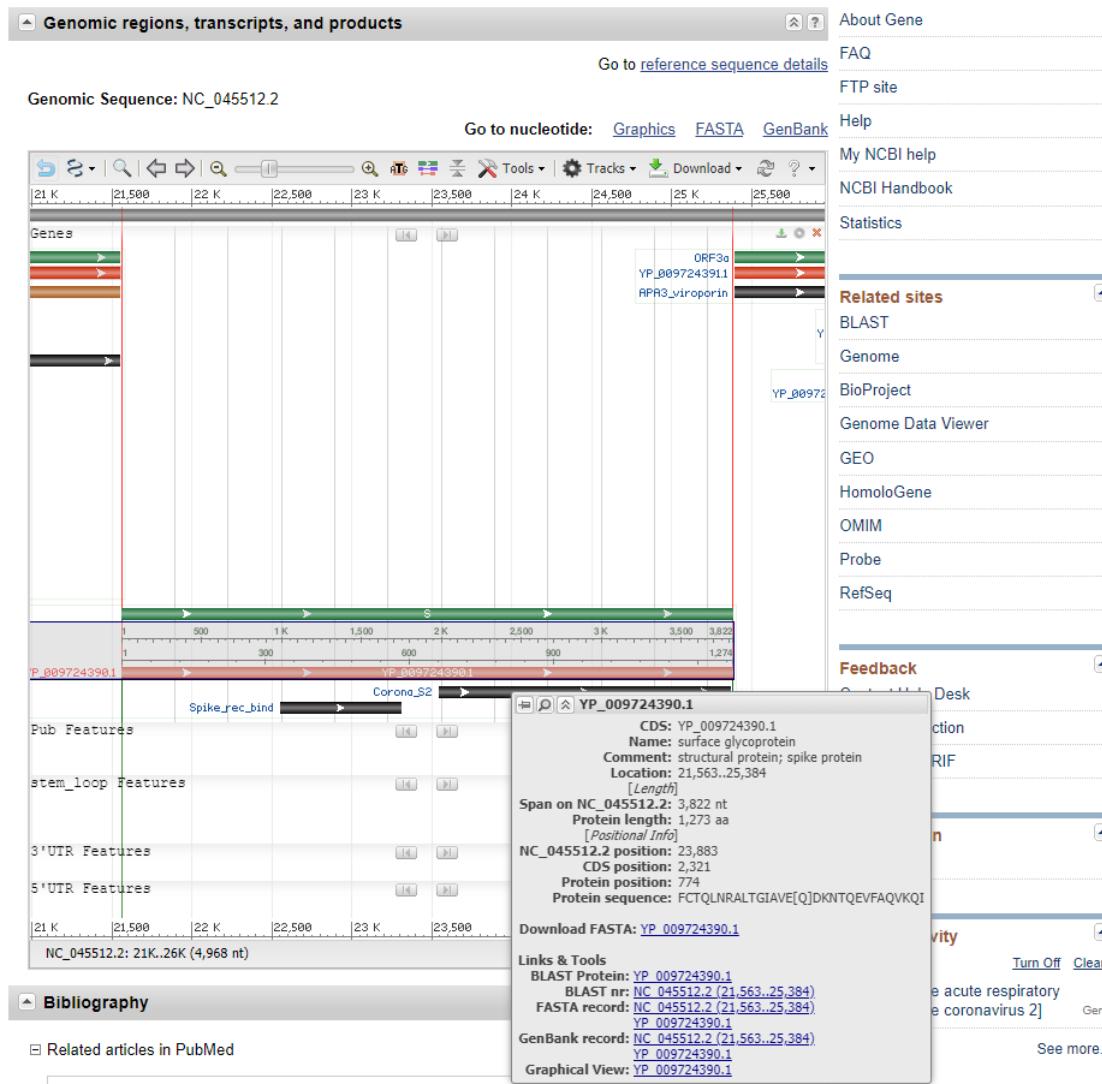


FIGURE 41 – Localisation du gène "spike" dans le génome du coronavirus SARS-CoV-2

D'après ce document, le gène se situe entre les nucléotides 21563 à 25384, ce qui est très proche des résultats précédents. Mais alors, comment expliquer cette différence entre notre algorithme et l'algorithme EMBOSS ?

La différence majeure réside en l'ajout d'un coût d'extension de gap. En effet, l'idée est de mettre un fort coût lors de la création d'un gap, et un très faible coût pour étendre ce gap.

```

#####
# Program: needle
# Rundate: Tue 5 Jan 2021 16:41:55
# Commandline: needle
#   -auto
#   -stdout
#   -asequence emboss_needle-I20210105-164137-0192-33463542-p2m.asequence
#   -bsequence emboss_needle-I20210105-164137-0192-33463542-p2m.bsequence
#   -datafile EDNAFULL
#   -gapopen 10.0
#   -gapextend 0.5
#   -endopen 10.0
#   -endextend 0.5
#   -aformat3 pair
#   -snucleotide1
#   -snucleotide2
# Align_format: pair
# Report_file: stdout
#####

=====
#
# Aligned_sequences: 2
# 1: EMBOSS_001
# 2: EMBOSS_001
# Matrix: EDNAFULL
# Gap_penalty: 10.0
# Extend_penalty: 0.5
#
# Length: 30036
# Identity: 3071/30036 (10.2%)
# Similarity: 3071/30036 (10.2%)
# Gaps: 25885/30036 (86.2%)
# Score: 9928.5

```

FIGURE 42 – Paramètres utilisés par l'algorithme de Needleman EMBOSS

En prenant comme dans l'exemple ci-dessus un coût de création de gap de 10 et un coût d'extension de gap de 0.5, on privilégie l'ajout de peu de gaps de grande taille, plutôt que de nombreux petits gaps. A l'exécution, cela augmente grandement la précision de l'alignement puisque mettre de nombreux petits gaps comme dans notre alignement couterait trop cher.

Ainsi, implémenter un coût d'extension de gap est une bonne piste d'amélioration pour notre algorithme, et permettrait d'obtenir des résultats beaucoup mieux exploitables pour une utilisation concrète en génomique.

### 3 Tests et performances

Dans cette partie, nous présentons les tests et performances de nos fonctions, et nous les comparons à la complexité théorique calculée dans la partie précédente.

#### 3.1 Fonctions utilitaires

Pour faciliter la mesure des performances et même le test de différentes fonctions, une poignée de fonctions utilitaires a été programmée. Parmi ces fonctions, il y a :

- `measure_single_call` : qui mesure un seul appel à une fonction, avec l'argument `args` et retourne un tuple qui contient la taille d'argument et le temps d'exécution en seconde.
- `func_performance` : qui prendra comme arguments une fonction, un tableau d'arguments et l'index ou les indices des arguments dont dépend l'exécution de la fonction. Il y a aussi des arguments moins pertinents comme celui qui permet de demander le trier par temps d'exécution ou par taille d'argument, ou bien un argument en booléen qui indiquent si nous voulons une figure ou non. Et enfin le `tick_spacing` qui spécifie l'espacement entre les graduations de l'axe des abscisses. La fonction renvoie une table qui contient un tuple dont la taille de l'argument et le temps d'exécution sont respectivement en seconde.
- `funcs_performance` : qui est plus ou moins similaire à `func_performance` sauf qu'il prend un tableau de fonctions. Le tri basé sur le temps d'exécution n'existe pas ici car il n'a pas de sens, puisque différentes fonctions ont des temps d'exécution différents, les arguments seront donc toujours triés dans l'ordre croissant. Les performances des différentes fonctions seront tracées dans le même graphique avec des couleurs différentes pour permettre une comparaison plus facile entre différentes versions de fonctions. Cette fonction renverra une table contenant la taille de l'argument, la paire de temps d'exécution pour chaque fonction respectivement.
- `plot_multi_graph` : Cette fonction a le même objectif que la fonction mentionnée ci-dessus mais au lieu de passer directement les fonctions et de mesurer leur temps d'exécution, elle prend les coordonnées précalculées et les trace. Cette fonction prend également les légendes pour chaque graphique.
- `arg_generator` : C'est peut-être la fonction la plus importante car elle peut être utilisée pour alimenter les fonctions avec de nombreux arguments générés pseudo-aléatoirement<sup>38</sup>. Cette fonction a beaucoup d'arguments et son comportement change légèrement en fonction de la valeur de ses arguments. Les arguments `start`, `N` et `stride` indiquent où la génération d'argument va commencer, où elle va se terminer et les étapes de la même manière qu'une boucle `for`. Le type indique si nous générerons un groupe de chaînes ou un tableau de nombres. Si la fonction génère une chaîne, l'argument `samples` spécifie quels caractères seront dans la chaîne, si l'argument `same_size` est défini sur `False`, la longueur des chaînes sera choisie au hasard entre `lower` et `upper`, sinon la longueur sera itérée de manière croissante. Si la fonction génère des nombres, les nombres `lower` et `upper` spécifieront la plage des nombres générés aléatoirement dans le tableau, et le tableau des nombres augmentera linéairement. Enfin, il y a `variant_arg_pos` et `static_args`, `variant_arg_pos` spécifiera l'emplacement des différents arguments, tandis que `static_arguments` sera la valeur des arguments qui ne change pas ces deux peuvent être utiles pour les fonctions dont la complexité ne dépend que de quelques arguments et pas de tous d'entre eux (comme l'algorithme de Needleman-Wunsch) connaissant ces deux arguments (le dernier peut être `None`), nous pouvons remplir tous les arguments de la fonction. La fonction renvoie donc à la fin de son execution une liste d'arguments générés pseudo-aléatoirement.

38. En raison de la nature déterministe des ordinateurs, les soi-disant nombres générés aléatoirement ne sont pas vraiment aléatoires. D'où l'appellation pseudo-aléatoire. (Nous utiliserons les deux termes de manière interchangeable dans le rapport, mais ce que nous voulons vraiment dire est pseudo-aléatoire)

- `func_performance_mt` : identique à `func_performance` mais utilise plusieurs processus. Elle applique la fonction donnée en paramètre aux arguments du tableau d'entrée, mais de manière simultanée. Elle trace ensuite le résultat sur un graphique.
- `funcs_performance_mt` : identique à `func_performance` mais utilise plusieurs processus. Elle applique `func_performance_mt` sur toutes les fonctions données en paramètre en parallèle puis trace le résultat.
- `funcs_performance_mt_v2` : identique à `funcs_performance_mt` mais le processus de parallélisation se fait à un niveau plus profond. Il appellera en série `func_performance_mt` pour chaque fonction, parallélisant ainsi l'exécution de la fonction qui nous intéresse sur ses arguments.

Vous pouvez peut-être remarquer le suffixe `mt`, dans les trois dernières fonctions, qui signifie multi-threading. Mais en réalité, ces fonctions utilisent le *multi-processing* et non le *multi-threading*. Avec Python, même lors de l'utilisation du *multi-threading*, il y a de grandes chances que le code soit exécuté de manière séquentielle en raison de l'interférence du GIL<sup>39</sup> avec l'exécution des *threads*. Au lieu de cela et pour surmonter tout le problème, plusieurs processeurs sont utilisés en utilisant le paquet `multiprocesseur` de python.

La raison derrière l'implémentation de telles fonctions en exécutions simultanées est que l'exécution séquentielle des tests de performance pourrait potentiellement prendre beaucoup de temps, en particulier avec des fonctions de complexité exponentielle comme `lev` et `needleman_all`. Et comme il n'y a pas de dépendances lors de l'exécution de ces fonctions<sup>40</sup> sur différents arguments, il y a donc une opportunité pour paralléliser l'exécution de l'ensemble des fonctions de test de performance et profiter du package `multiprocessing` que Python fourni par défaut.

Nous avons aussi remarqué que `arg_generator`, qui était principalement utilisé et implémenté pour les tests de performances, peut être légèrement modifié et utilisé pour réaliser des tests générés pseudo-aléatoirement. Donc, un tas de ces tests ont été mis en oeuvre partout où nous le pouvions dans nos fichiers de tests.

## 3.2 Les tests des fonctions

Tous les tests ont été écrits en utilisant `pytest` et en s'appuyant sur les principes *Right-BICEP*. Les librairies `BioPython` et `NumPy` ont été utilisées aussi en testant avec ses fonctions prédéfinies. Toutefois, pour des tests plus simples et objectifs, des tests aléatoires ont été réalisés pour certaines fonctions avec la fonction `arg_generator` défini dans le fichier `src/performance.py`.

### 3.2.1 Les fonctions de statistiques

Les tests de toutes les fonctions calculant les statisques ont été faits en parcourant toutes les situations possibles (dans le cadre du projet) ; des cas de listes vides ont été ajoutés, les fonctions renvoient `None`. De plus des tests avec la fonction `arg_generator` ont été faits, pour choisir des nombres pseudo-aléatoires. Par ailleurs, lors du test de `quartile` avec le module `NumPy`, le choix de l'interpolation était nécessaire. Au début, elle n'était pas précisée, donc par défaut l'interpolation était linéaire, or celle de la fonction `quartile` ne l'est pas donc, il y a eu des échecs, puis après modification de l'interpolation<sup>41</sup> les tests sont passés avec succès.

39. Le Global Interpreter Lock est un verrou (lock) que l'interpréteur utilise afin d'être thread-safe sur le comptage des références du ramasse-miette.

40. Et du coup l'absence de nécessité d'utiliser un mécanisme de verrouillage *Locking Mechanism*

41. en `lower` et `higher`

### 3.2.2 Les fonctions codons

Des différents tests ont été écrits pour les trois versions de `codons`. Pour la fonction `codons`, des tests codés en dur ont été réalisés, ainsi que deux autres fonctions `test_codons_bio_seq` et `test_codons_gen`. En effet, `test_codons_bio_seq` test la fonction `codons` en utilisant le module `Seq` prédéfini de la librairie `biopython`. Toutefois, à travers la fonction, on a adapté les résultats de `Seq` de sorte que ça soit compatible aux résultats de la fonction `codons` qui est conforme à la documentation comme expliqué dans la partie 2.2. Quant à `test_codons_gen`, elle est conforme à `test_codons_bio_seq` mais en générant pseudo-aléatoirement les arguments pour les tests. C'est le cas aussi pour `test_codonv2_gen` et `test_codonv3_gen` qui sont conformes respectivement à `test_codon_v2` et `test_codon_v3_bio_seq`. Il faut noté aussi qu'à travers les tests on a constaté que la fonction `codon_v2` est la version la plus conforme à la fonction `translate()` du module `Seq` de `biopython`.

### 3.2.3 Les fonctions : distance de Levenshtein

Comme dans l'implémentation de l'algorithme de Levenshtein trois versions ont été réalisées, toutes les versions ont été alors testées. Deux méthodes de tests ont été proposées. La première consiste à créer un tableau (liste des listes) où on entre manuellement des arguments à tester sous la forme de : [chaine1<sup>42</sup>, chaine2<sup>43</sup>, entier<sup>44</sup>]. Ensuite, on effectue les tests par les fonctions `test_rec` et `test_dp` en testant respectivement `lev_rec` et `lev_dp`. Quant à la fonction `test_lev_total`, elle teste tous les fonctions à savoir `lev_rec`, `lev_dp` et `lev` en les comparant entre elles. Enfin, la deuxième méthode de test, décrite par la fonction `test_gen_lev`, consiste à générer les arguments aléatoirement<sup>45</sup> tout en comparant les trois versions entre elles.

### 3.2.4 Les fonctions : algorithme de Needleman-Wunsch

Le test des fonctions de Needleman n'était pas aussi simple que d'autres fonctions. De nombreux problèmes ont été rencontrés lors de la mise en œuvre. Tout d'abord, nous voulions comparer les résultats de nos fonctions à ceux de BioPython. Nous avons commencé par utiliser le module BioPython `pairwise2.align`, ce qui nous a posé problème car la fonction ne renvoie pas tous les alignements possibles. Au début, nous pensions que c'était un bug et nous l'avons signalé sur la page BioPython sur GitHub<sup>46</sup>. Un contributeur appelé *mdehoon* nous a suggéré d'utiliser le module le plus récent `PairwiseAligner` de BioPython, nous l'avons donc choisi pour nos tests. Un autre contributeur appelé Markus Piotrowski a expliqué que le module `pairwise2.align` n'est pas défectueux mais il ne renvoie pas tous les alignements possibles car certains d'entre eux sont redondants et n'ont pas d'utilisation significative dans la vie réelle ; et car la sortie d'alignement est limitée à 1 000 pour les fonctions de `pairwise2.align`. Les tests dans l'ensemble nous ont permis de découvrir quelques bugs qui ont été corrigés par la suite. Certains de ces bugs étaient mineurs, comme les bugs qui viennent d'ufait que python effectue une copie superficielle au lieu d'une copie profonde par défaut pour les tableaux. D'autres étaient plus fondamentaux et liés à la logique de l'algorithme lui-même. Pour tester toutes les fonctions de Needleman-Wunsch avec leurs différentes versions, nous avons utilisé quelques tests écrits à la main avec un tas de différents tests générés pseudo-aléatoirement. La quantité de tests générés peut être contrôlée avec les paramètres déclarées en haut des fichiers de test<sup>47</sup>. Ces paramètres sont utiles pour effectuer plus de tests, les réduire ou les désactiver complètement.

42. La première séquence pour le calcul de la distance de Levenshtein

43. La deuxième séquence pour le calcul de la distance de Levenshtein

44. La distance de Levenshtein entre les deux séquences (le résultat attendu)

45. par la fonction `arg_generator`

46. <https://github.com/biopython/biopython/issues/3453>

47. les autres fichiers de test contenant des tests générés de manière pseudo-aléatoire suivent la même tendance

ment, ce qui devient rapidement pratique, d'autant plus que certaines fonctions prennent beaucoup de temps à s'exécuter.

### 3.3 Les tests de performance

#### 3.3.1 Les fonctions de statistiques

Les tests de performances sont réalisés sur des listes d'éléments générés pseudo-aléatoirement<sup>48</sup>, elles sont de taille entre 1 à 100 000.

**moyenne** a une complexité théorique linéaire  $\Theta(n)$  pour la fonction **moyenne**. Selon le test de performances, voir figure 43, l'allure de la courbe du temps d'exécution de **moyenne** semble plutôt linéaire, comme le prévoit la complexité théorique.

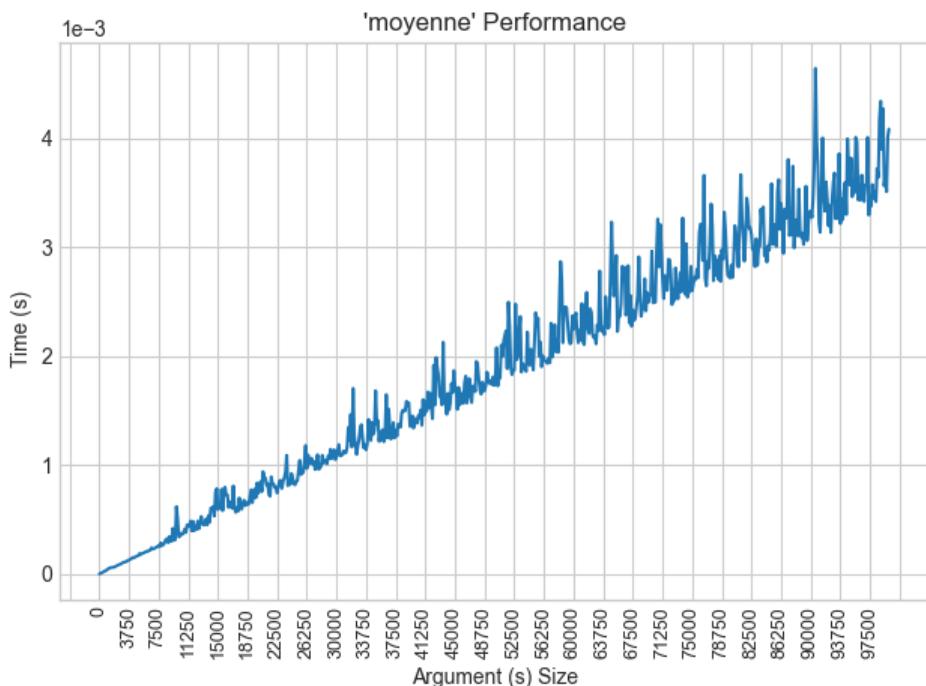


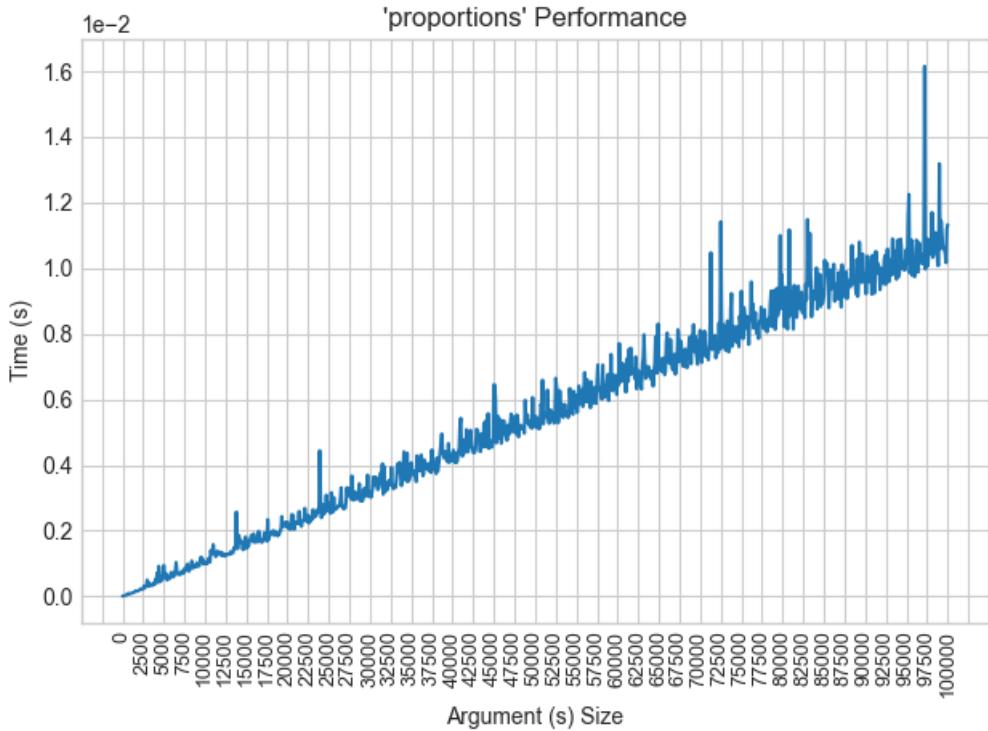
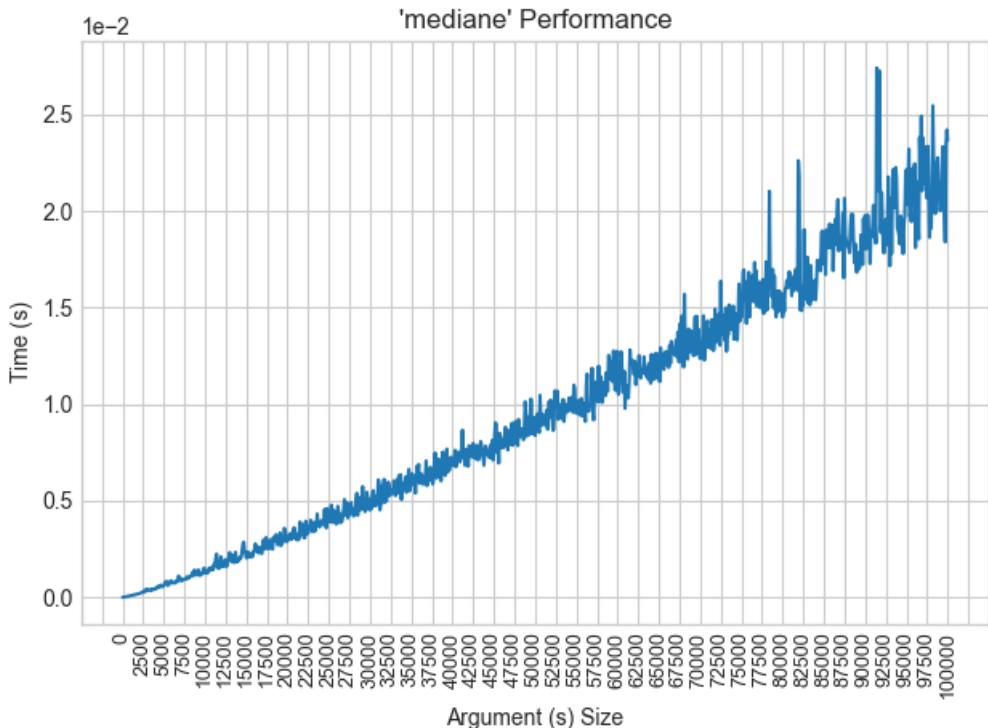
FIGURE 43 – Test de performance de la fonction **moyenne** sur des listes de taille 1 à 100 000

**proportion** a une complexité théorique linéaire  $\Theta(n)$ . D'après le test de performance, voir figure 44, le temps de réponse de **proportions** semble assez linéaire, comme l'indique la complexité théorique.

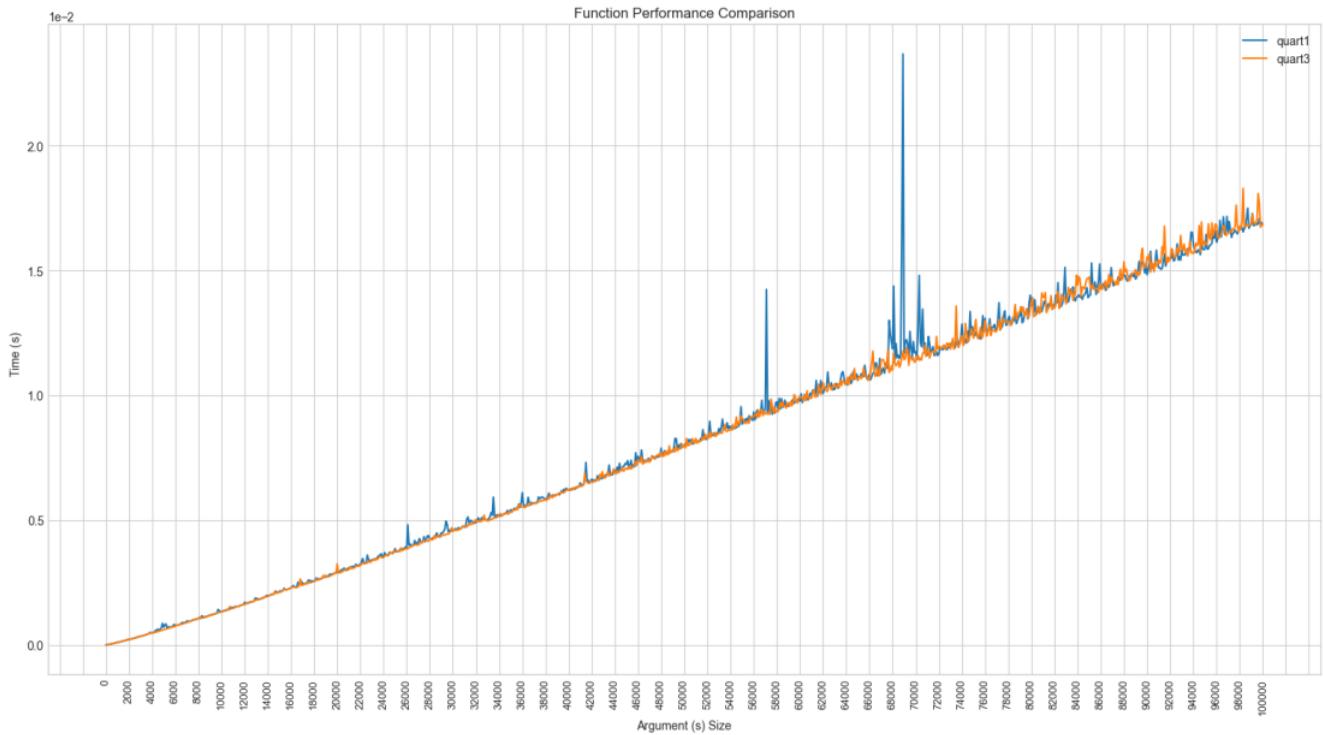
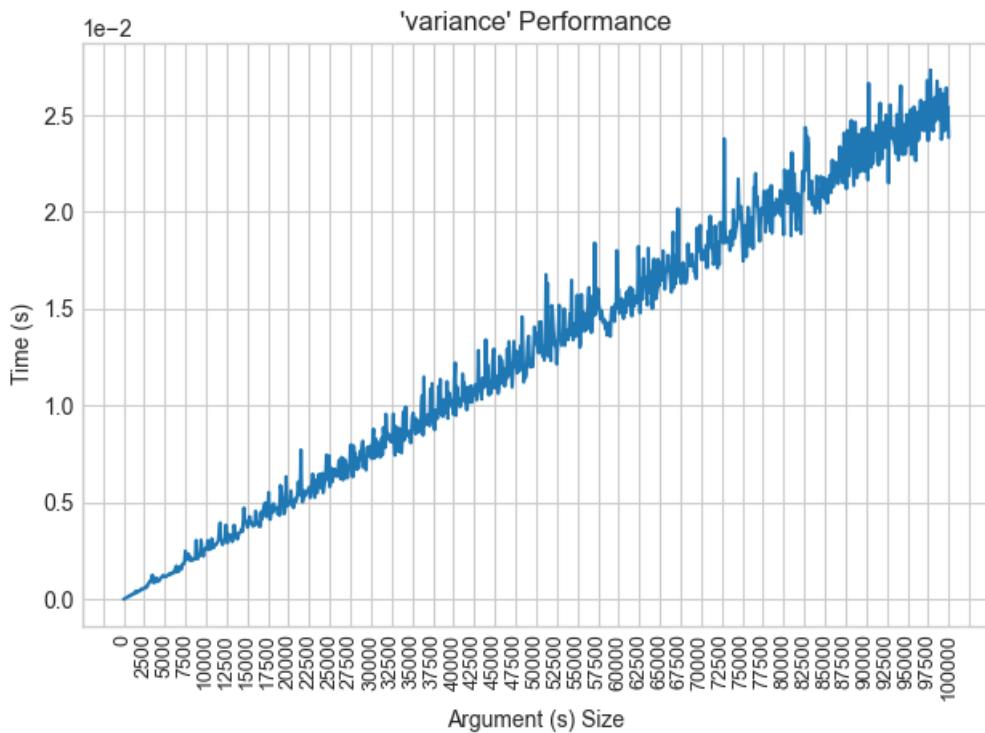
**mediane** a une complexité théorique quasi-linéaire  $\Theta(n \log(n))$  dans le pire des cas, et linéaire ( $\Theta(n)$ ) dans le meilleur des cas. Selon le test sur la figure 45, le temps de réponse de **mediane** semble quasi-linéaire (linéaire légèrement arondi), cela ressemble à la complexité théorique.

**quartile** a la même complexité que la fonction **mediane**  $\Theta(n \log(n))$  dans le pire des cas, et linéaire  $\Theta(n)$  dans le meilleur des cas). D'après les tests de performance sur la figure 46, le temps de réponse de **quartile** semble quasi-linéaire (linéaire légèrement arondi), ce qui est comparable à la complexité théorique.

48. avec la fonction `arg_generator`

FIGURE 44 – Test de performance de la fonction `proportions` sur des listes de taille 1 à 100 000FIGURE 45 – Test de performance de la fonction `mediane` sur des listes de taille 1 à 100 000

`variance` a une complexité théorique linéaire  $\Theta(n)$ . Selon le test de performance, voir figure 47, le temps de réponse de `variance` est linéaire, comme le prévoit la complexité théorique.

FIGURE 46 – Test de performance de la fonction `quartile` sur des listes de taille 1 à 100 000FIGURE 47 – Test de performance de la fonction `variance` sur des listes de taille 1 à 100000

`ecart_type` se comporte comme la fonction `variance`, elle a une complexité théorique linéaire  $\Theta(n)$ . Selon les tests de performances, voir figure 48, le temps de réponse de `ecart_type` est linéaire, comme la complexité théorique de la `variance`.

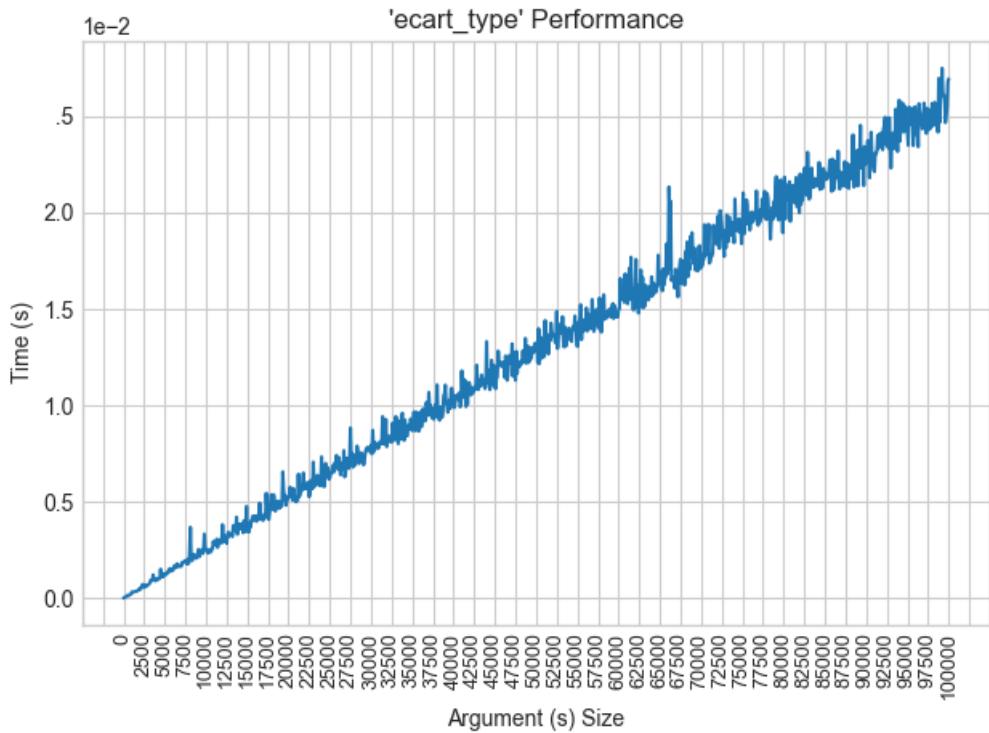


FIGURE 48 – Test de performance de la fonction `ecart_type` sur des listes de taille 1 à 100 000

`intervalle_interquartile` appelle deux fois la fonction `quartile` pour une soustraction des deux, elle a une complexité théorique quasi-linéaire en  $\Theta(n \log(n))$ . D'après nos tests de performances, voir figure 49, l'allure de la courbe du temps d'exécution est quasi-liénaira.

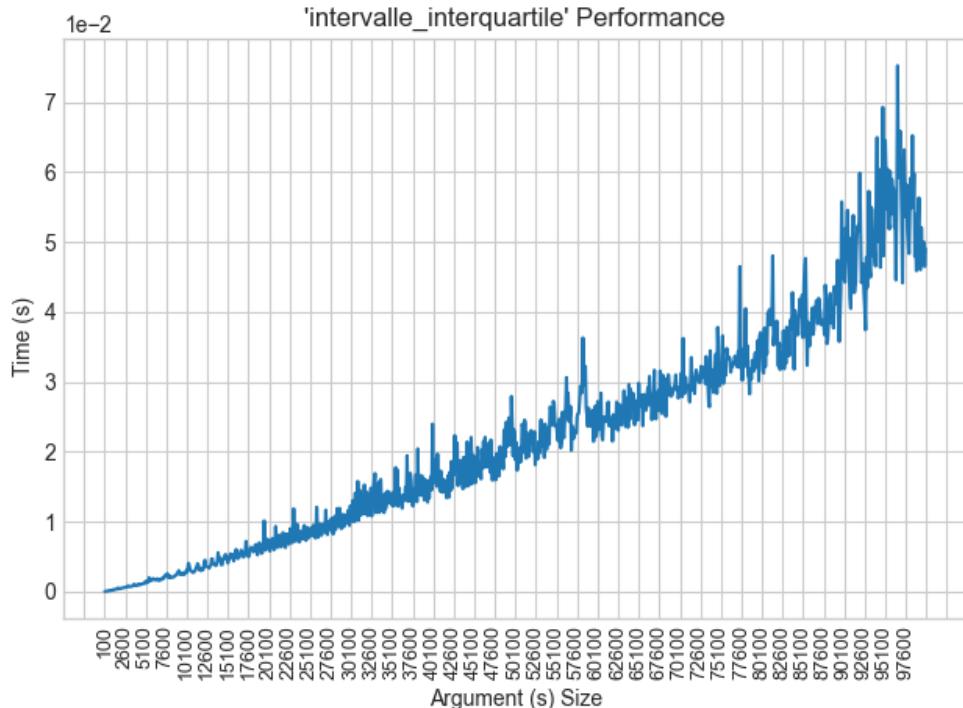


FIGURE 49 – Test de performance de la fonction `intervalle_interquartile` sur des listes de taille 1 à 100 000

### 3.3.2 Les fonctions codons

Les tests de performances des fonctions `codons` sont réalisés sur des chaînes de caractères générées aléatoirement, elles sont de taille entre 1 et 9 000. Les trois fonctions `codons`, `codon_v2` et `codon_v3` ont une complexité théorique linéaire  $\Theta(n)$ . Selon le test de performances, voir figure 50, l'allure des courbes du temps d'exécution des fonctions semblent plutôt linéaire, ce qui est conforme à la complexité théorique.

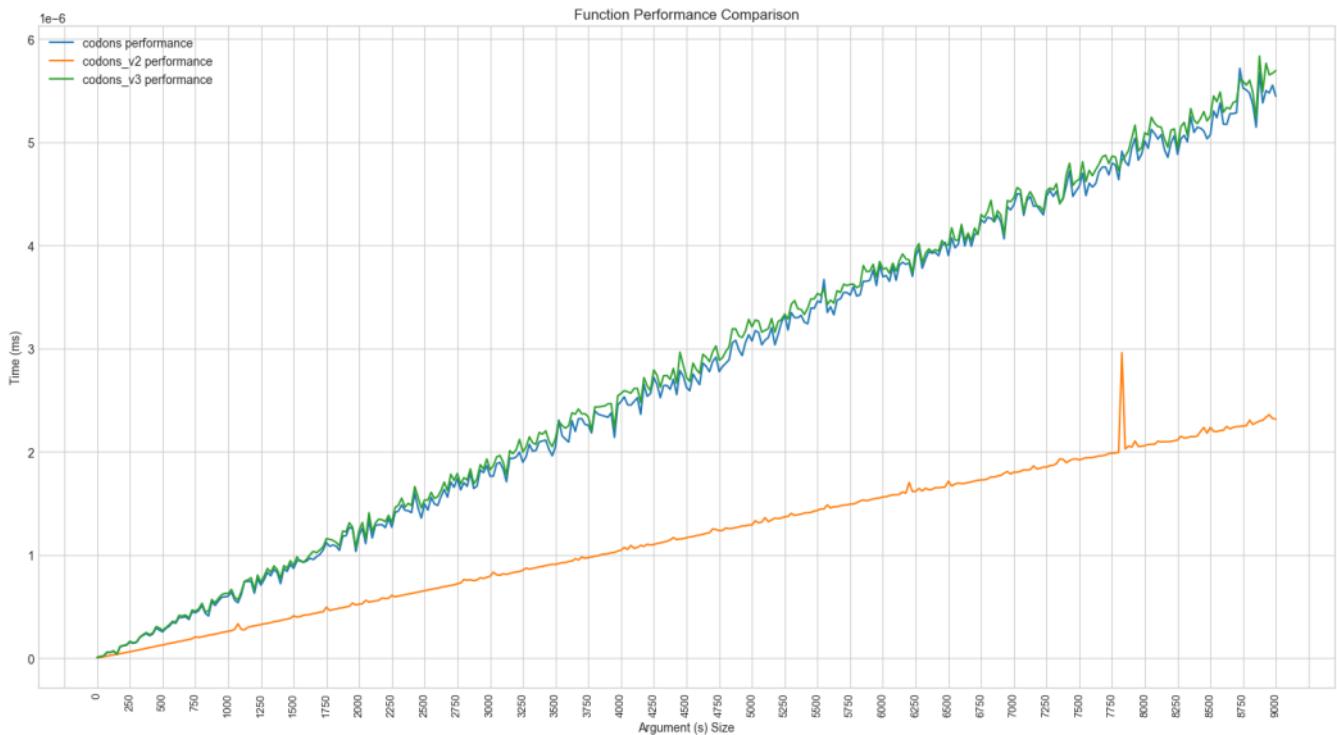


FIGURE 50 – Test de performance des fonctions `codons`, `codons_v2` et `codons_v3`

### 3.3.3 Les fonctions : distance de Levenshtein

Nous avons discuté dans la section 2.3 que la complexité théorique de l'algorithme de Levenshtein est  $O(n.m)$  pour `lev_dp` et `lev` et  $O(3^{n+m})$  pour `lev_rec` où  $n$ ,  $m$  sont respectivement la longueur de la première et la deuxième séquence. Nous allons vérifier ces résultats théoriques et comparer les différentes versions de l'implémentation Levenshtein. Il est important de noter que pour faciliter la mesure, le tracé des graphiques, et pour avoir des résultats cohérents, nous avons exécuté nos tests de performances sur des arguments de même taille à chaque fois. En utilisant cette simplification les complexités en  $O(n.m)$  deviendront des complexités en  $O(n^2)$ . Puisque nous savons déjà de quoi dépend la complexité de l'algorithme, cette simplification n'aura pas d'impact sur les résultats ni sur la vérification de la complexité hypothétique.

**lev** Il s'agit de la version itérative de l'implémentation de l'algorithme de Levenshtein utilisant la programmation dynamique. Nous voyons, en effet, que cette version est la plus rapide parmi toutes les autres implémentations. On remarque également la même forme de courbure que `lev_dp` mais avec des coefficients différents, ce qui vérifie notre hypothèse<sup>49</sup>. La forme d'une courbe ressemble à une fonction  $n$  au carré, ceci est compatible avec nos attentes, voir figure 51.

49. qui dit que les deux ont la même complexité de  $O(n.m)$

`lev_dp` Il s'agit de la version récursive de l'algorithme de Levenshtein utilisant la programmation dynamique. On voit que cette version est plus rapide que `lev_rec` (qui est entièrement récursive sans programmation dynamique) mais toujours plus lente que la version itérative, car il y a un surcoût dû à la création de l'environnement de la fonction à chaque appel récursif, voir figure 51.

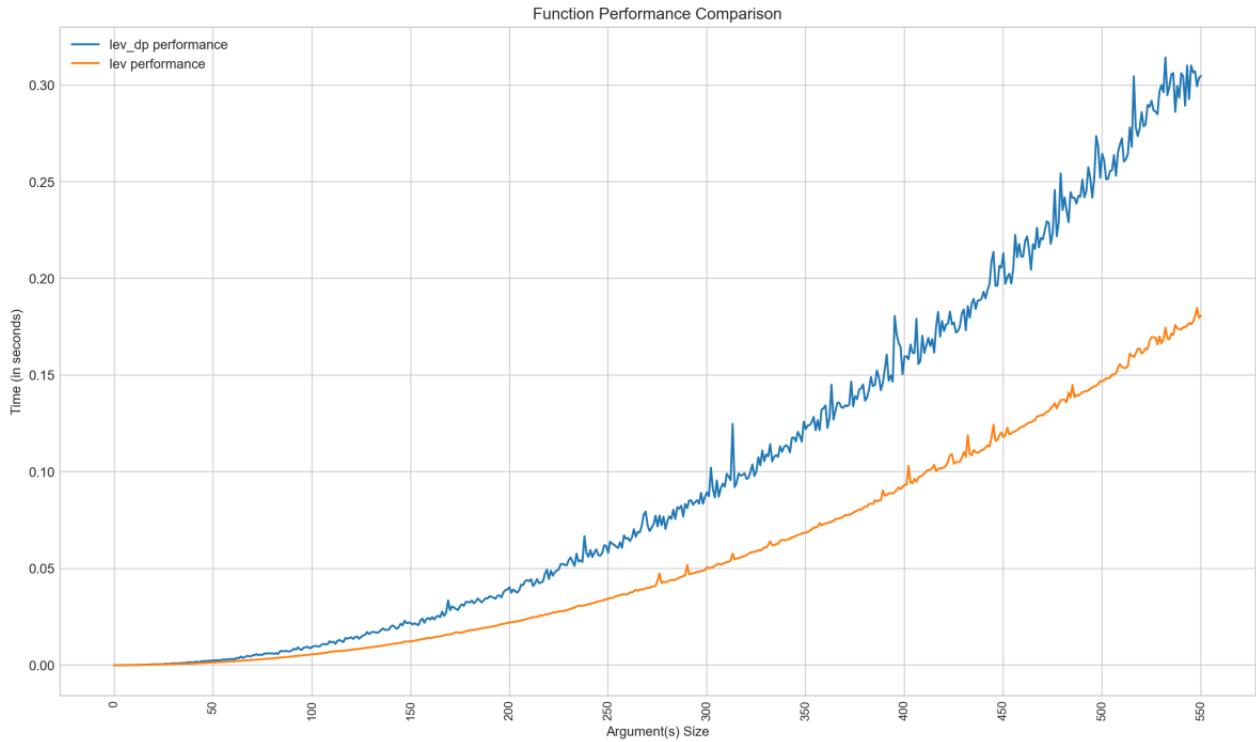
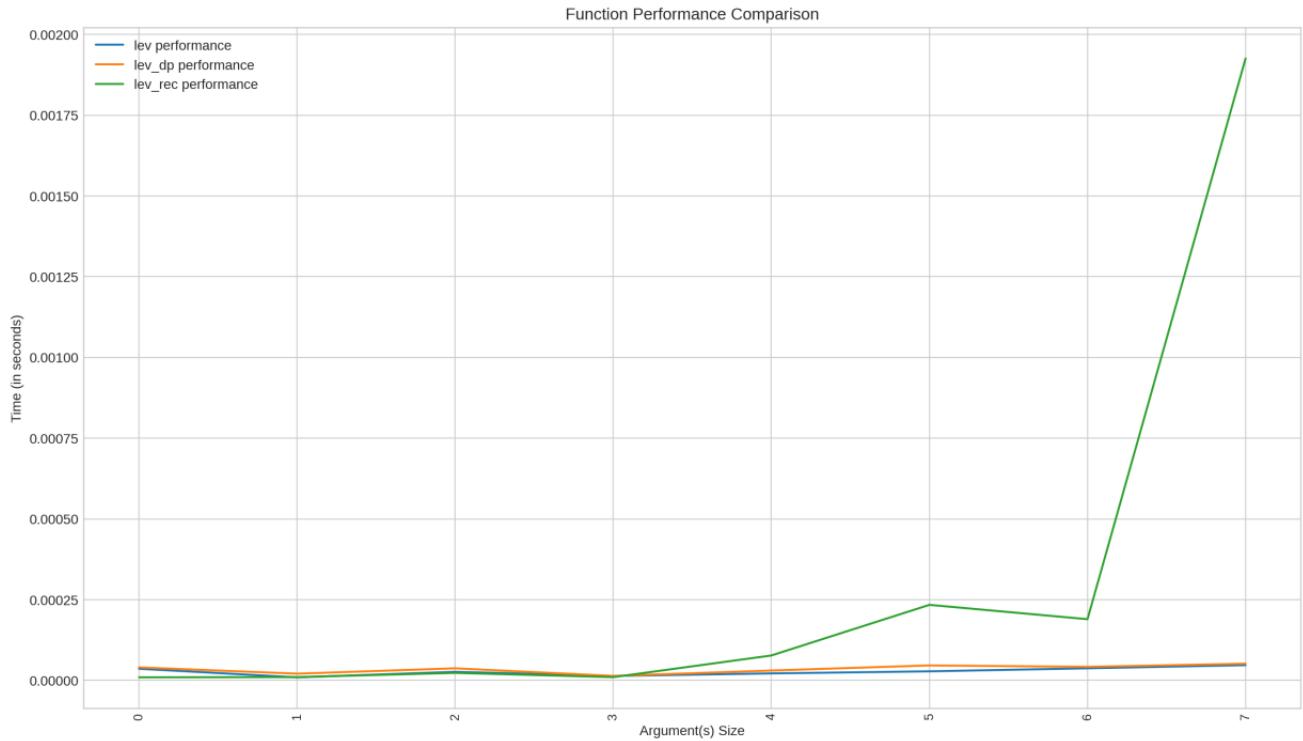


FIGURE 51 – Test de performance des fonctions `lev` et `lev_dp`

`lev_rec` Est la fonction la plus lente de toutes et aussi la fonction qui a le plus grand taux de croissance, qui est en fait une croissance exponentielle, voir figure 52. Puisqu'il n'utilise aucune technique de mémorisation et recalcule même les résultats déjà calculés. Ce qui explique sa complexité exponentielle.

On peut conclure que l'utilisation de la programmation dynamique optimisera grandement l'implémentation récursive naïve qui permet de réduire sa complexité de  $O(3^{n+m})$  à  $O(n.m)$ .

FIGURE 52 – Test de performance des fonctions `lev`, `lev_dp` et `lev_rec`

### 3.3.4 Les fonctions : algorithme de Needleman-Wunsch

Nous avons discuté dans la section 2.4 que la complexité théorique de `needleman` qui produit un seul alignement est  $\Theta(n.m)$  et `needleman_all` a une complexité de  $O(e^{n+m})$  où  $n,m$  est respectivement la longueur de la première et de la deuxième séquence. Nous confirmerons non seulement ces affirmations, mais nous les comparerons également au résultats de performance du module `PairwiseAligner` de BioPython.

`needleman` La forme du temps d'exécution de `needleman` ressemble à la courbure d'une fonction  $n$  au carré, voir figure 53.

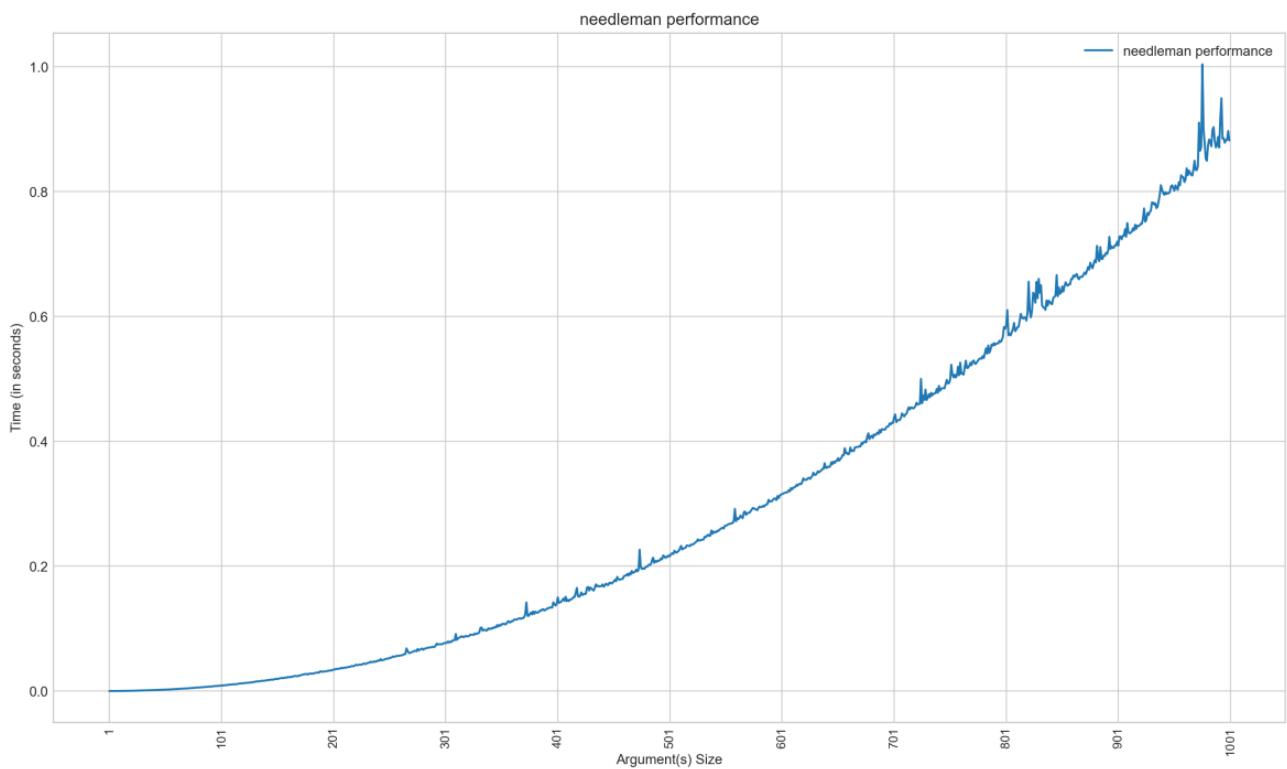


FIGURE 53 – Test de performance des fonctions `needleman`

Ce qui est normal puisque nous avons suivi la même stratégie utilisée pour mesurer les performances de Levenshtein, en prenant des arguments de même taille pour simplifier les mesures et le tracé des graphes. On peut donc en conclure que la complexité théorique que nous avons calculée est correcte ( $O(n.m)$ ).

**needleman\_all et PairwiseAligner** En regardant sur le graphique 54 pour le pire des cas, nous remarquons que `needleman_all` et `PairwiseAligner` croissent de façon exponentielle. Les tests de performance ont été réalisés sur des échantillons aléatoires de nucléotides allant de 1 à 7, avec un coût nul pour toutes sortes d'opérations (c'est-à-dire le *match*, *mismatch* et le *gap*). nous pouvons conclure que la borne supérieure que nous avons théoriquement calculée était en fait exacte.

À partir du graphique 55 du temps d'exécution en moyen des cas, nous voyons que la courbe de `needleman_all` ressemble à un  $n^2$  qui est cohérent avec sa complexité moyenne. Cependant, nous remarquons également beaucoup de fluctuations, principalement des sauts vers des valeurs très élevées. Ces sauts peuvent être expliqués par la complexité dans le pire des cas qui est de l'ordre de  $O(e^{n+m})$ . Ces tests utilisent des matrices de similarité et il n'y a rien de spécial à cela, nous pouvons plutôt choisir la méthode standard et nous obtiendrons des résultats similaires. Nous avons cependant choisi cette version pour varier.

Dans l'ensemble, il y a encore une marge d'amélioration pour la fonction `needleman_all`. Tout d'abord, nous pouvons éliminer tous les alignements redondants. Deuxièmement, nous pouvons exécuter en parallèle l'étape de retraçage qui est l'étape la plus lourde, avec la plus grande complexité (autrement dit le *bottleneck*<sup>50</sup>). Troisièmement, nous pouvons coder la fonction entière dans un langage de programmation de bas niveau comme C ou C++ qui s'exécute beaucoup plus rapidement que Python, puis fournir une interface pour pouvoir l'appeler via Python (c'est ce que BioPython fait pour la plupart de ses fonctions).

50. le goulot d'étranglement

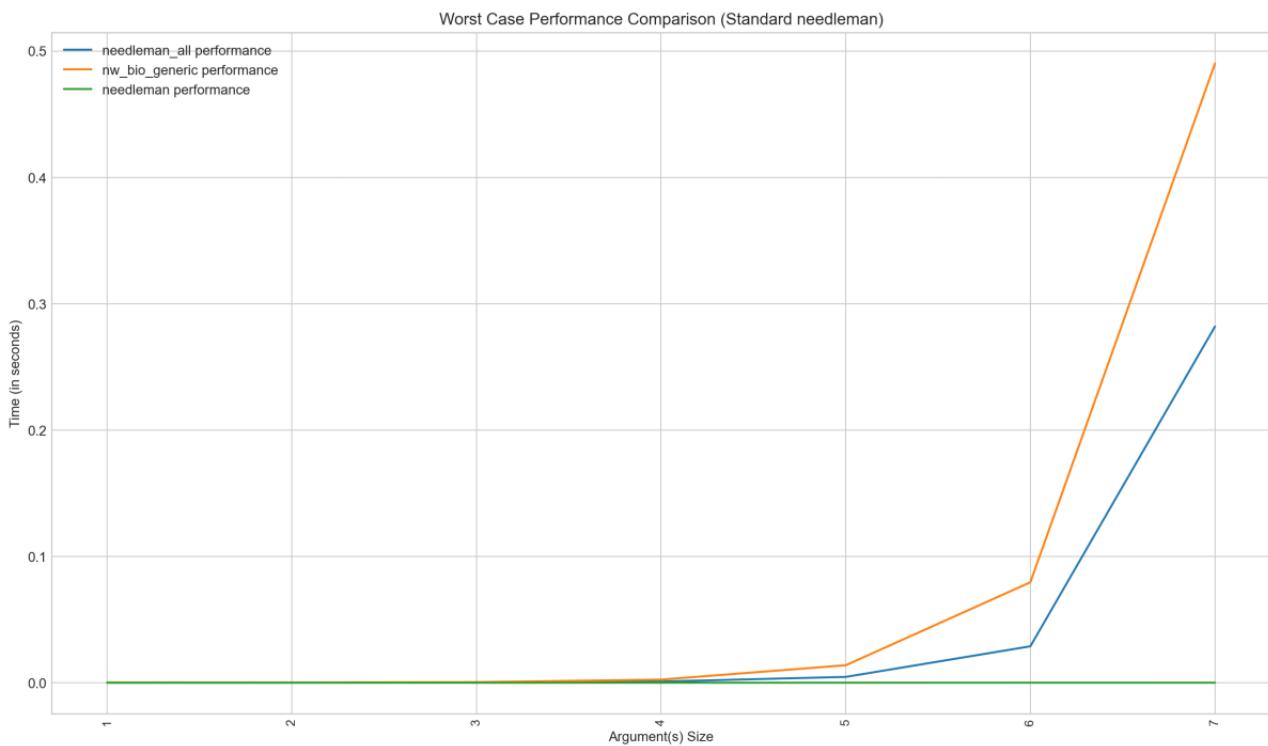


FIGURE 54 – Test de performance des fonctions `needleman`, `needleman_all` et `nw_bio_generic` (cette dernière est la fonction de `biopython`)

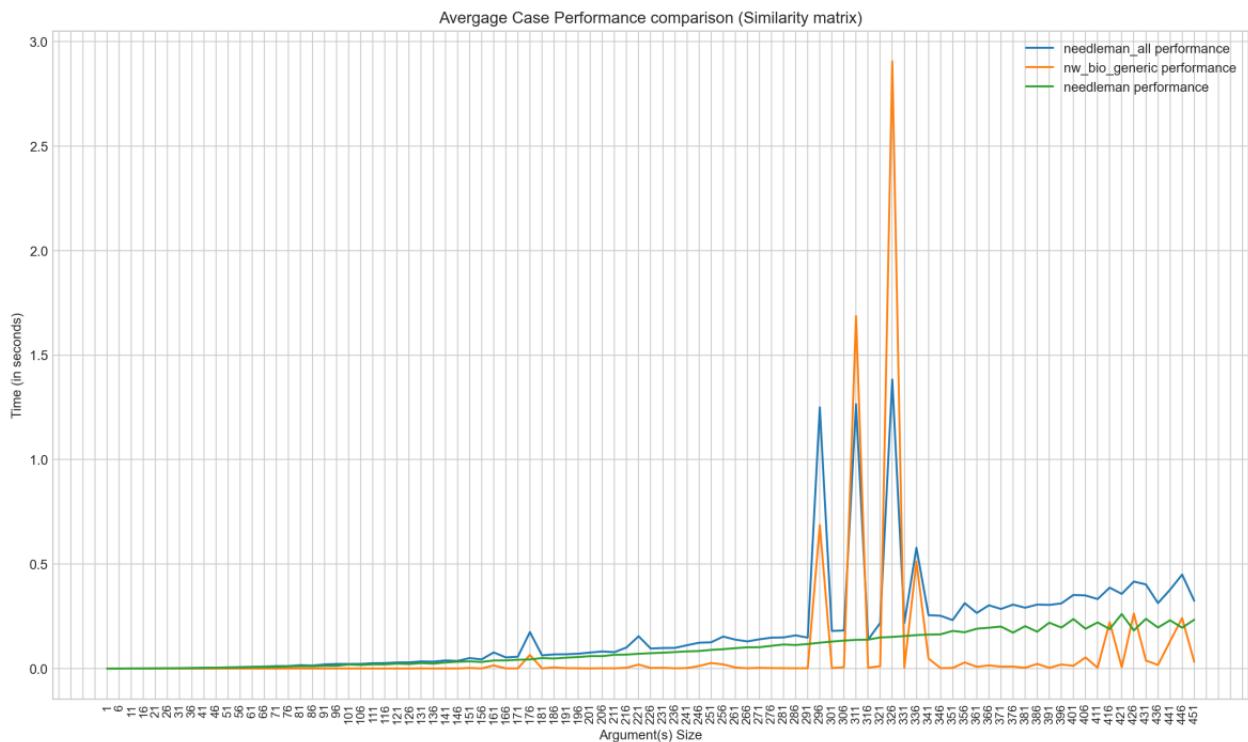


FIGURE 55 – Test de performance des fonctions `needleman`, `needleman_all` et `nw_bio_generic` (cette dernière est la fonction de `biopython`) cas moyen

## 4 Gestion de projet

Cette partie est consacrée à la présentation de notre gestion de projet. L'équipe s'est inspiré du cadre méthodologique SCRUM, pour l'organisation de l'équipe et du projet, voir la table 5, la méthode SMART a été utilisé pour le découpage des tâches, voir la table 6 ; avant le début du projet une analyse des risques en utilisant la matrice SWOT a été effectuée, voir la figure 56.

### 4.1 Équipe de projet

Ce projet a été réalisé par le groupe 13 dont les membres sont :

- Mohamed Omar CHIDA, le leader développeur du SCRUM, il s'assure du découpage des tâches, s'occupe de régler tous les soucis informatique au sein du projet, et il se charge de trancher lors des débats ;
- Mathis DUMAS, le développeur côté documentaire, il s'occupe l'exactitude des notions abordées, vérifie les sources de celles-ci et s'occupe des graphes d'analyse du rapport ;
- Chaima TOUNSI OMEZZINE, le développeur code reviewer, elle s'assure de l'utilité de chaque fonction, la correspondance avec le sujet du projet (les cahiers de charges), et vérifie les tests de celles-ci ;
- Céline ZHANG, le développeur organisateur, elle s'occupe de l'organisation des réunions, du déroulement des réunions, de la répartition des tâches à la fin de celles-ci, de l'avancée du rapport, et elle se charge de rédiger les comptes rendus de réunions.

Chaque membre est développeur, donc participe à la conception et à l'écriture des codes.

Membre de l'équipe 13	Rôles ou charges
Mohamed-Omar CHIDA	Leader
Mathis DUMAS	Documentation
Chaima TOUNSI OMEZZINE	Reviewer
Céline ZHANG	Organisateur

TABLE 5 – La répartition des rôles

### 4.2 Analyse du projet

#### 4.2.1 Définition des objectifs

Les objectifs ont été définis en se basant sur la méthode SMART comme indiqué sur la table 6 :

	Critère	Indicateur
S	Spécifique	Objectif clair, précis et sans ambiguïté
M	Mesurable	Objectif quantifié permettant de mesurer l'état d'avancement
A	Ambitieux et Atteignable	Objectif représentant un défi atteignable et non démotivant
R	Réaliste	Objectif envisageable et suffisamment motivant
T	Temporellement défini	Objectif défini et délimité dans le temps

TABLE 6 – La méthode SMART

#### 4.2.2 Analyse des risques : Matrice SWOT

Nous avons évalué les risques du projet en utilisant la matrice SWOT sur la figure 56.

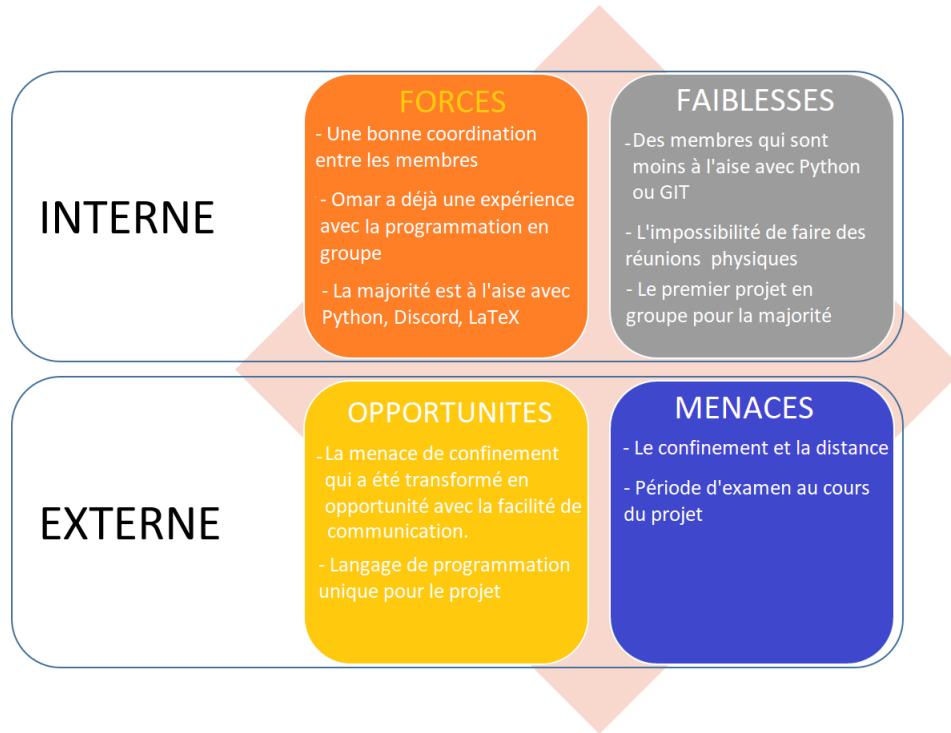


FIGURE 56 – La matrice SWOT du projet

### 4.3 Organisation du projet

#### 4.3.1 Durée

Le projet a commencé 18 Octobre 2020 et s'est terminé le 5 Janvier 2020, pour une durée de 80 jours. Cette version du rapport a été terminée le 5 Janvier 2021.

#### 4.3.2 Le cadre méthodologique SCRUM

Pour une meilleure organisation, on a choisi d'adapter une gestion de projet agile avec SCRUM. On a estimé que l'adaptation collective et rapide sera plus productive et plus constructive pour tous les membres de l'équipe.

Par conséquent, on s'est basé sur les 3 piliers fondamentaux de SCRUM :

- Transparence (visibilité concrète sur la situation)
- Inspection (détection des écarts par rapport aux objectifs)
- Adaptation (réctification de ces écarts par rapport aux objectifs)

Conformément à ce cadre, un ordonnancement du product backlog a été effectué, au début. On a divisé les tâches sur des *sprints* fixés afin de pouvoir concevoir, réaliser et tester les nouvelles fonctionnalités au fur et à mesure. En effet, pour chaque *sprint* on fixe un objectif à court terme et on se lance dans la réalisation. Une fois cet objectif atteint, on discute et on s'adapte à la situation, à travers les réunions hebdomadaires.

Toutes ces organisations ont été mis en oeuvre dans le tableau Trello<sup>51</sup> comme illustré dans la figure 57.

51. <https://trello.com/b/N92bj53G/projet-grp13>

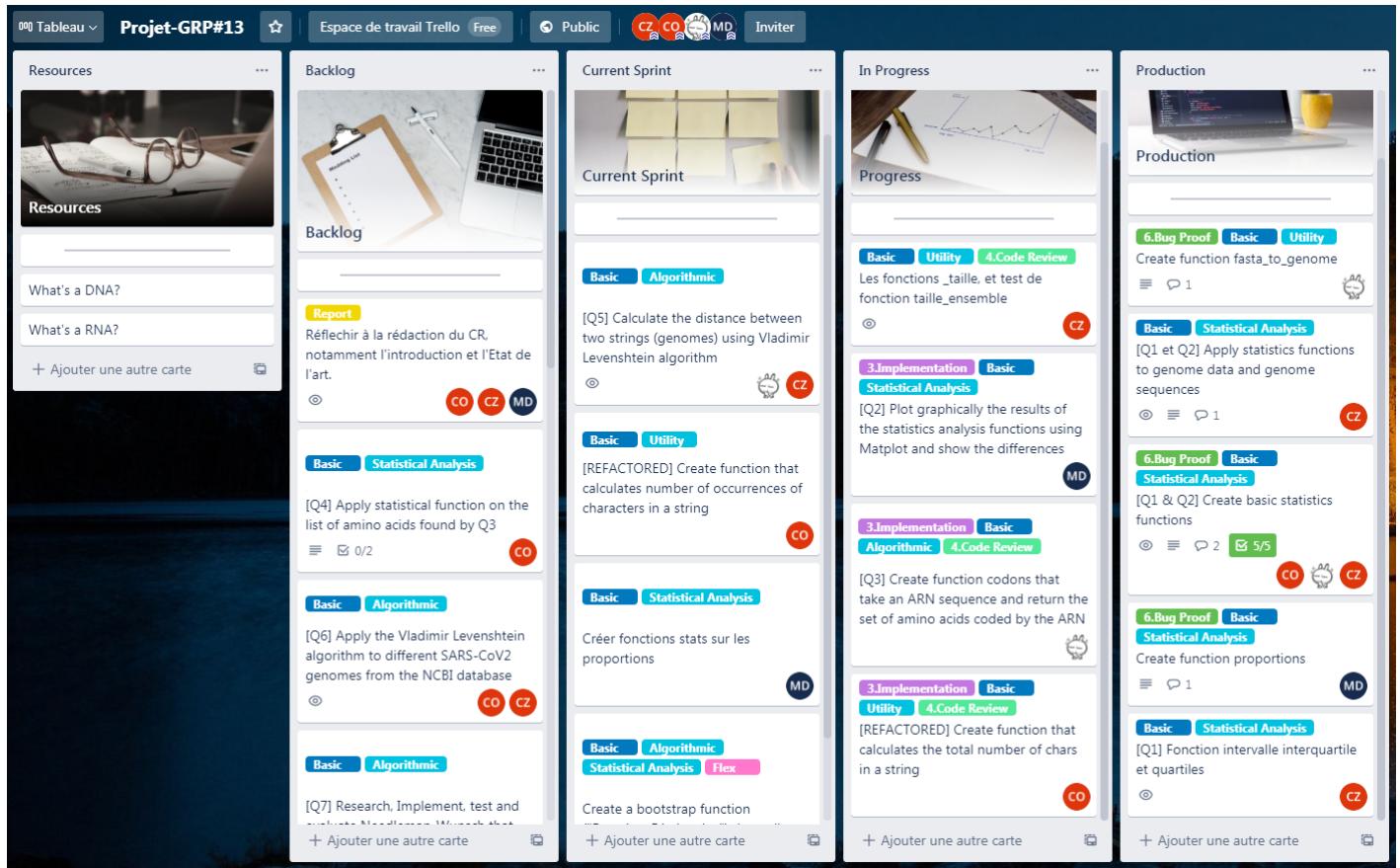


FIGURE 57 – L'organisation du projet sur Trello

L'organisation du tableau Trello était comme suit :

- **Resources** : c'est les documentations qu'il faut faire.
- **Backlog** : la liste de tâches prioritées.
- **Current Sprint** : le Sprint au cours de réalisation.
- **In Progress** : les tâches en cours de réalisation.
- **Production** : les tâches finies.

Pour chaque tâche il y a un ensemble d'étiquettes par lesquelles on pourrait avoir une idée sur la tâche et son avancement. On distingue principalement des étiquettes qui précisent la nature de la tâches comme :

- **Algorithmic** : si la tâche consiste à écrire un algorithme ou implémenter une fonction. Par exemple, l'algorithme de Needleman ou la distance de Levenshtein.
- **Statistical Analysis** : est attribuée pour toutes les tâches en relation avec l'analyse descriptive.
- **Utility** : est attribuée aux tâches qui consistent à implémenter des fonctions utiles pour les fonctions de base. Par exemples, la fonction `fasta_to_genome`.
- **Benchmark** : est attribuée aux tâches de performances.
- **Report** : si la tâche est en relation avec la rédaction du rapport.
- **Presentation** : si la tâche est en relation avec la réalisation de la présentation.

Des autres étiquettes qui permettent de suivre l'avancement des tâches :

1. **Research** : la phase de documentation
2. **Design/Conception** : la phase de la conception de l'algorithme.

3. **Implémentation** : la phase de l'implémentation de l'algorithme.
4. **Code Review** : une étiquette qui est attribuée à la tâche après l'implémentation pour que les autres développeurs passent vérifier le code.
5. **Testing** : la phase des tests.
6. **Bug Proof** : quand une tâche est finie, on attribue cette étiquette et on la glisse dans la colonne Production.

Finalement, il y a aussi des étiquettes qui précisent le caractère de la tâche :

- **Basic** : des tâches basiques qui sont prioritaire.
- **Flex** : des tâches supplémentaires non demandées par l'énoncé.
- **Issue** : est attribuée à la tâche si le développeur chargé par celle-ci a des problèmes.

## 4.4 Outils de travail

### 4.4.1 Communication

Les principales outils de communication sont : Discord, Messenger, GitHub

**Discord** a été utilisé pour communiquer en messages, pour organiser des réunions, montrer les codes, streamer des explications, et éventuellement envoyer des documents.

**Messenger** est principalement utilisé pour la communication des réunions, l'échange des disponibilités et la discussion des particularités des tâches lors des *sprint* en dehors des réunions.

**GitHub** a permis de communiquer avec les contributeurs de la bibliothèque `biopython` afin de pouvoir comprendre l'utilisation des modules (`Aligner`, `pairwise2`).

### 4.4.2 IDE

Selon les préférences de chaque membre, plusieurs outils de programmation ont été utilisés, les IDE : PyCharm, Visual Studio Code et IDLE.

### 4.4.3 Partage du travail

Tout au long du projet, nous avons utilisé le dépôt **GitLab** fournit par l'école, **Google Drive** (pour les débuts), et **workupload**<sup>52</sup> pour le partage de gros documents (comme les échantillons de séquences du génome). Nous avons créé plusieurs répertoires pour organiser nos fichiers :

- **src** pour les fichiers de code.
- **tests** pour les fichiers test.
- **app** pour les applications des algorithmes.
- **rappor**t pour enregistrer les résultats des applications.
- **genome** où on ajoute les fichiers .fasta pour les applications.

---

52. <https://workupload.com/>

#### 4.4.4 Rédaction du rapport

Le rapport a été rédigé sur Overleaf permettant aux membres de modifier leurs parties simultanément, cela facilite la compilation du rapport (éitant les soucis de packages ou versions). Un répertoire `rapport` a été mis en place aussi sur le dépôt Gitlab dans lequel est rassemblé l'ensemble des résultats des applications.

### 4.5 Les réunions de projet

Les réunions ont été essentielles pour planifier les *sprints*, régler les problèmes rencontrés. Nous avons eu 12 réunions de groupe sur **Discord** (de durée totale de 39 heures) qui sont listées dans le tableau 7, pour lesquelles des comptes rendus de réunion ont été écrites, ils se trouvent dans les annexes. Par ailleurs quelques *stand-up-meeting* ont été fait en début de projet et pendant les périodes de partiels à l'école **Télécom Nancy**.

Date	Durée	Lieu
10 Novembre 2020	5h	Discord
14 Novembre 2020	5h	Discord
21 Novembre 2020	3h	Discord
28 Novembre 2020	3h	Discord
5 Décembre 2020	2h	Discord
20 Décembre 2020	3h	Discord
23 Décembre 2020	3h	Discord
26 Décembre 2020	3h	Discord
29 Décembre 2020	3h	Discord
2 Janvier 2021	3h	Discord
4 Janvier 2021	4h	Discord
5 Janvier 2021	2h	Discord

TABLE 7 – Les réunions de groupe

## Conclusion

Pour conclure le projet, l'implémentation des fonctions statistiques ont permis d'analyser les séquences du génome SARS-CoV-2 et d'avoir une idée globale des caractéristiques de celui-ci. De plus, pour une étude plus avancée, la fonction `codons` a été réalisée afin d'identifier les sous-séquences d'acides aminés dans les séquences d'ARNm, cela a permis de faire des analyses statistiques sur celles-ci en plus des analyses sur les séquences de nucléotides. La taille d'une séquence de nucléotides est d'environ 29812( $\pm 43$ ), avec pour nombres de nucléotides 8901( $\pm 20$ ) Adénine, 9580( $\pm 20$ ) Uracile, 5849( $\pm 11$ ) Guanine, 5474( $\pm 11$ ) Cytosine, pour proportions respectives 29.9%, 32.1%, 19.6%, 18.4% ; la taille d'une séquence d'acides aminés est d'environ 9612( $\pm 11$ ), le nombre de chaque acide aminé se trouve sur la figure 29, et leur proportion se trouve sur le diagramme circulaire 31. Afin de mesurer les différences entre les séquences d'acides aminés qui représentent les protéines, la fonction calculant la distance de Levenshtein a été conçue, elle permet de connaître la distance d'édition entre deux séquences représentées par deux chaînes de caractères. Dans l'intention d'étudier les alignements globaux entre deux séquences du génome, des fonctions (`needleman`, `needleman_all`, `nw_verbose`) calculant l'algorithme de Needleman-Wunsch ont été (codées), elles nous permettent respectivement d'avoir un alignement global avec son score, tous les alignements avec leur score, et l'affichage animé du retraçage en couleur du chemin dans la matrice pour obtenir un alignement global.

En vue de vérifier le bon fonctionnement des fonctions, plusieurs tests exhaustifs ont été effectués, en parcourant toutes les situations possibles, notamment les cas limites, et en ajoutant des tests pseudo-aléatoires. Ces tests ont permis de corriger des coquilles, des bugs, des incohérences. (De plus) des mesures de performances ont été réalisées dans le but de comparer le temps d'exécution des fonctions avec leur complexité théorique. Elles ont mis en évidence l'importance des méthodes d'implémentation pour réduire la complexité de la fonction. En effet, plusieurs problèmes, comme `stack overflow` et le temps d'exécution pour des entrées de grandes tailles, ont été soulignés, certaines fonctions avaient une complexité exponentielle sans l'utilisation de la programmation dynamique.

Dans l'ensemble, l'équipe estime avoir bien traité le sujet dans sa totalité, bien qu'il y ait eu quelques ambiguïtés sur les travaux demandés, de ce fait plusieurs versions ont été réalisées. L'équipe se félicite du travail sérieux qu'elle a fourni pour ce projet, les membres sont satisfaits de leurs travaux et de ceux des coéquipiers.

## Bilan global du projet d'équipe

### Le point sur le projet

Travaux demandés	Travaux réalisés
Fonctions d'analyses statistiques	moyenne, mediane, quartile, proportions, variance, ecart_type, intervalle_interquartile
Application des fonctions d'analyses sur les séquences de nucléotides	Applications sur les séquences de nucléotides, call_stats, perform_all_stats, call_stats_prop, perform_all_stats_prop, call_stat_taille_genome, perform_all_stats_taille, fasta_to_genome Illustrations graphiques, interprétation : Des fonctions d'illustrations graphiques sont faites dans le fichier <code>stat_plot.py</code>
Fonction codons	codons, codons_v2, codons_v3, start_to_stop, try_AUGC
Application des fonctions d'analyses sur les séquences de codons	Applications sur les séquences de codons, illustrations graphiques, interprétations codons_échantillon, bank_sequences, stats_plot.py
Fonction distance de Levenshtein	lev, lev_rec, lec_dp
Application de la fonction distance de Levenshtein sur les séquences de codons	Applications sur les séquences de codons, interprétations lev, codons, codons_v2, codons_v3
Fonctions algorithme Needleman-Wunsch	needleman, needleman_all, nw_bio, nw_bio_mat, nw_bio_generic
Afficher les opérations de l'algorithme Needleman-Wunsch	needleman_seq.py : nw_verbose affichage d'un tableau avec le retraçage pour l'alignement
Calcul de complexité	Les fonctions de stats, codons, lev et needleman, etc. ont leur complexité
Tests de fonctions	dossier tests avec pytest : module BioSeq, module PairwiseAligner, module NumPy
Mesures de performances	En utilisant le module timeit et les fonctions de performance.py les tests de performances se trouvent dans le fichier perf_funcs.py

TABLE 8 – Le bilan du projet

Points positifs	Expérience en travail d'équipe Application des notions vues en cours dans des situations concrètes Amélioration des compétences de programmation en python Application des connaissances en gestion de projet Progression des compétences de tests de fonctions et de tests de performances Bases en python Harmonie de l'équipe Participation active de chacun des membres Réunions régulières Accessibilité des ressources en ligne
Points négatifs	Sujet ambigu, certains aspects ne sont pas précisé Cela a causé un problème de perte de temps Plusieurs fonctions sont faites en réponse à une seule demande Coupure du projet dû à la période d'examens Confinement
Difficultés	Le travail à distance et l'impossibilité du travail en groupe en physique Des nouvelles notions de biologies qu'il faut comprendre pour pouvoir coder L'utilisation du biopython Rédaction du rapport
Améliorations possibles	Limiter les chiffres significatifs des résultats en analyse statistique Optimiser le stockage mémoire pour la fonction lev Augmenter la vitesse d'exécution en écrivant en C Réduire la complexité des needleman Rédaction du rapport

TABLE 9 – Le bilan d'équipe

### Le point sur l'équipe

Thèmes	M. O. CHIDA	M. DUMAS	C. TOUNSI OMEZZINE	C. ZHANG
Documentation	8h	12h	11h	12h
Conception	14h	5h	7h	8h
Implémentation	28h	14h	10h	11h
Code Review	12h	2h	8h	3h
Tests et performances	18h	5h	7h	9h
Rédaction des documents	8h	25h	25h	29h
TOTAL	88h	63h	68h	72h

TABLE 10 – Le temps moyen consacré au projet de chaque membre

## Annexes

### Les déclarations sur l'honneur de non-plagiat

#### Déclaration sur l'honneur de non-plagiat

**Je, soussigné,**

**Nom, prénom : CHIDA, Mohamed Omar**

**Élève ingénieur inscrit en 1ère année à TELECOM Nancy**

**Numéro de carte étudiante : 31730598**

**Année universitaire : 2020 - 2021**

**Auteur, en collaboration avec DUMAS Mathis, TOUNSI OMEZZINE Chaima et ZHANG Céline, du rapport :**

#### Algorithme pour l'analyse du SARS-COV2

Je déclare sur l'honneur que ce rapport est le fruit d'un travail personnel et que je n'ai ni contre-fait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier, texte ou code informatique, afin de la faire passer pour mienne.

Je certifie donc que le travail rendu est un travail original et que les sources utilisées, notamment pour les formulations, les idées, les documentations, les raisonnements, les analyses, les schémas ou autres créations ont été mentionnées conformément aux usages en vigueur.

Je suis conscient que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université et qu'il peut être sévèrement sanctionné.

Fait à Nancy, le 03/01/2021

Signature de l'étudiant :



## Déclaration sur l'honneur de non-plagiat

**Je, soussigné,**

**Nom, prénom : DUMAS, Mathis**

**Élève ingénieur inscrit en 1ère année à TELECOM Nancy**

**Numéro de carte étudiante : 32012997**

**Année universitaire : 2020 - 2021**

**Auteur, en collaboration avec CHIDA Mohamed Omar, TOUNSI OMEZZINE**

**Chaima et ZHANG Céline, du rapport :**

### Algorithmme pour l'analyse du SARS-COV2

Je déclare sur l'honneur que ce rapport est le fruit d'un travail personnel et que je n'ai ni contre-fait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier, texte ou code informatique, afin de la faire passer pour mienne.

Je certifie donc que le travail rendu est un travail original et que les sources utilisées, notamment pour les formulations, les idées, les documentations, les raisonnement, les analyses, les schémas ou autres créations ont été mentionnées conformément aux usages en vigueur.

Je suis conscient que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université et qu'il peut être sévèrement sanctionné.

Fait à Nancy, le 03/01/2021

Signature de l'étudiant :



## Déclaration sur l'honneur de non-plagiat

**Je, soussignée,**

**Nom, prénom : TOUNSI OMEZZINE, Chaima**

**Élève ingénieur inscrit en 1ère année à TELECOM Nancy**

**Numéro de carte étudiante : 32025001**

**Année universitaire : 2020 - 2021**

**Auteur, en collaboration avec CHIDA Mohamed Omar, DUMAS Mathis et ZHANG**

**Céline, du rapport :**

### Algorithmme pour l'analyse du SARS-COV2

Je déclare sur l'honneur que ce rapport est le fruit d'un travail personnel et que je n'ai ni contre-fait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier, texte ou code informatique, afin de la faire passer pour mienne.

Je certifie donc que le travail rendu est un travail original et que les sources utilisées, notamment pour les formulations, les idées, les documentations, les raisonnement, les analyses, les schémas ou autres créations ont été mentionnées conformément aux usages en vigueur.

Je suis consciente que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université et qu'il peut être sévèrement sanctionné.

Fait à Nancy, le 03/01/2021

Signature de l'étudiant :



## Déclaration sur l'honneur de non-plagiat

**Je, soussignée,**

**Nom, prénom : ZHANG, Céline**

**Élève ingénieur inscrit en 1ère année à TELECOM Nancy**

**Numéro de carte étudiante : 32024925**

**Année universitaire : 2020 - 2021**

**Auteur, en collaboration avec CHIDA Mohamed Omar, DUMAS Mathis et TOUNSI OMEZZINE Chaima, du rapport :**

### Algorithmme pour l'analyse du SARS-COV2

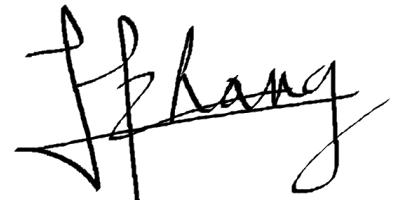
Je déclare sur l'honneur que ce rapport est le fruit d'un travail personnel et que je n'ai ni contre-fait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier, texte ou code informatique, afin de la faire passer pour mienne.

Je certifie donc que le travail rendu est un travail original et que les sources utilisées, notamment pour les formulations, les idées, les documentations, les raisonnement, les analyses, les schémas ou autres créations ont été mentionnées conformément aux usages en vigueur.

Je suis consciente que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université et qu'il peut être sévèrement sanctionné.

Fait à Nancy, le 03/01/2021

Signature de l'étudiant :



## Trello

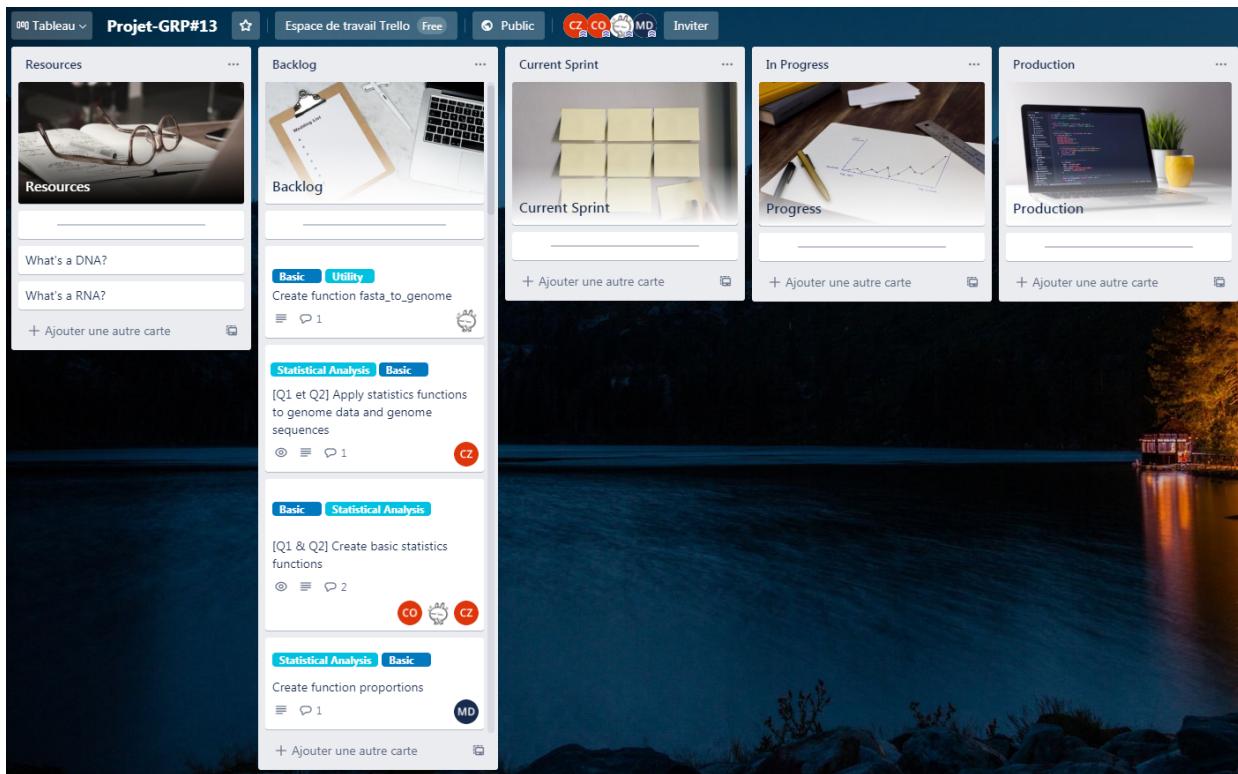


FIGURE 58 – L'organisation du Trello en début de projet

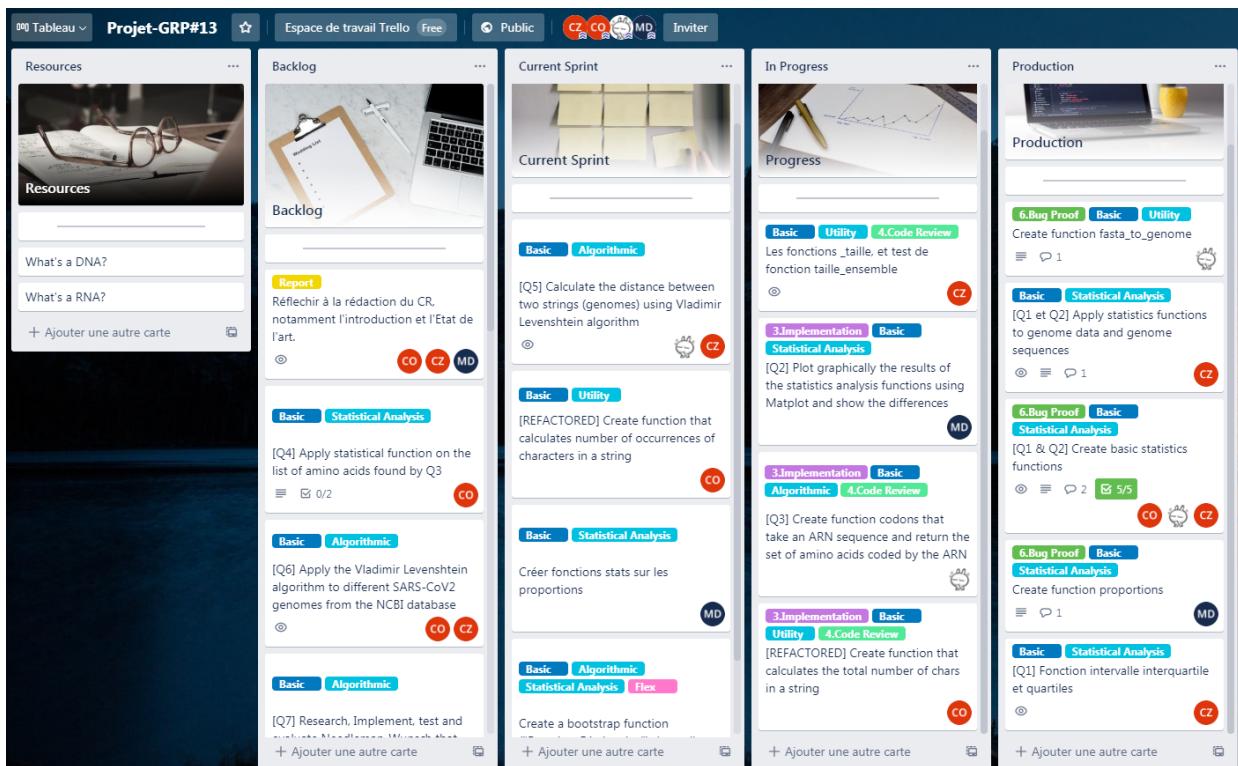


FIGURE 59 – L'organisation du Trello en milieu de projet

21. <https://trello.com/b/N92bj53G/projet-grp13>

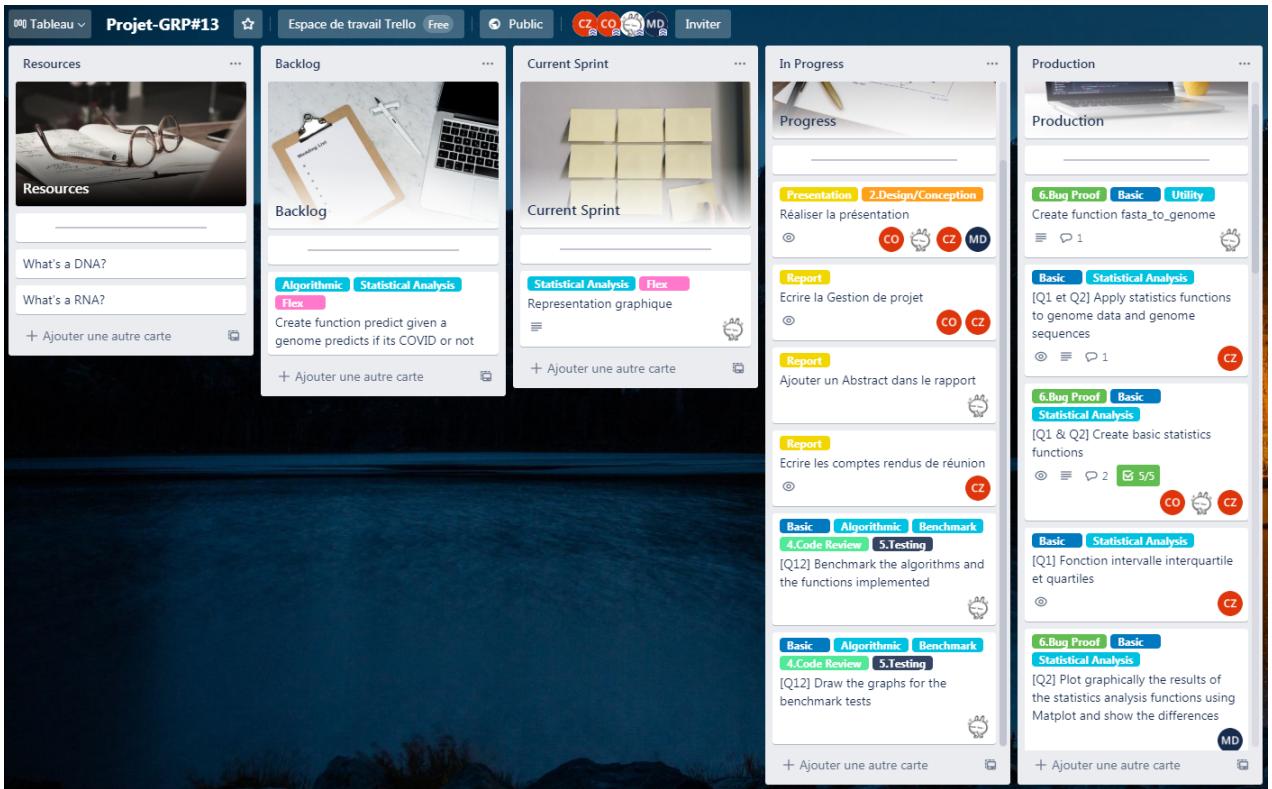


FIGURE 60 – L'organisation du Trello en fin de projet

## Comptes rendus de réunions

### Réunion d'équipe du 10 novembre 2020

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		5h	Discord

#### Ordre du jour

1. Mise à niveau générale sur les connaissances du sujet
2. Organisation du projet
3. Installation des outils nécessaires au projet
4. Découpage du sujet en tâches
5. Planification des prochaines réunions

#### Mise à niveau générale sur les connaissances du sujet

Nous avons mis en commun nos connaissances concernant le sujet, partagé nos documentations, et expliqué les notions ambiguës, suivi d'une présentation de nos préférences ou nos points forts respectives.

#### Organisation du projet

Nous avons parlé du sens de déroulement général du projet (quand faire les documentations, les premières lignes, quand aborder quelles parties, quand commencer la rédaction de rapport, etc.), en abordant également la gestion de projet (les stratégies, les rôles, voir le tableau 5, les facilités, les difficultés par la matrice de SWOT, voir figure 56), et puis nous avons parlé de la structure de nos dépôts, nos dossiers, nos codes, notre rapport (en définissant les styles, les normes, etc.). Nous adoptons la gestion SCRUM, qui utilise la méthode du *sprint*<sup>53</sup> pour réaliser les travaux.

Membre de l'équipe 13	Rôles ou charges
Mohamed-Omar CHIDA	leader
Mathis DUMAS	documentation
Chaima TOUNSI OMEZZINE	reviewer
Céline ZHANG	organisateur

#### Installation des outils nécessaires au projet

La présentation des outils utilisables et les outils nécessaires est primordiale pour commencer à travailler. Nous avons donc présenté les outils à disposition, créé nos dossiers et fichiers dans le dépôt, installé les logiciels.

53. Méthode d'organisation de travail d'équipe qui consiste à se fixer des tâches pour un cycle (1 à 2 semaines) et de les finir avant la fin de celle-ci.

## Découpage du sujet en tâches

En utilisant le tableau de bord Trello<sup>54</sup> nous avons découpé le cahier de charge en tâches nécessaires à la réalisation du livrable final. Ce qui nous permettra aussi de surveiller l'avancer de chacun sur ses tâches.

## Planification des prochaines réunions

Nous décidons de faire en général une réunion par semaine (flexible suivant les disponibilités), qui sera tous les samedis à 18h00 sur Discord, bien sûr il peut y avoir des sortes de stand-up-meeting en appel pour régler les soucis ou prendre des nouvelles (le tableau Trello permet aussi de laisser des messages pour indiquer les problèmes et demander de l'aide).

## ***TO-DO LIST***

- Se documenter (si nécessaire) et réfléchir à la première partie du sujet concernant les notions de descriptions du SARS-COV2
- Commencer à écrire les lignes de codes pour la première partie
- Envoyer un mail M. Festor pour demander la langue à utiliser pour les commentaires

*Prochaine réunion : 14/11/2020*

---

54. Outil qui nous permet de planifier en ligne des activités. 59

## Réunion d'équipe du 14 novembre 2020

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		5h	Discord

### Ordre du jour

1. Avancement de chacun sur le projet (première partie)
2. Résolution des difficultés rencontrées
3. Optimisation des solutions proposées
4. Vérification simpliste des tests
5. Planification du *sprint*

### Avancement de chacun sur le projet

Nous avons discuté de l'avancement de chacun sur la première partie du sujet, en mettant en commun nos idées, nos découvertes et nos codes.

**Omar CHIDA** a organisé le tableau de bord Trello 58, a mis une structure pour accéder au fichier par import pour les tests, il a proposé la première version de la fonction `fasta_to_genome` qui se trouve dans `utility.py`.

**Mathis DUMAS** a écrit plusieurs premières versions des fonctions de `utility.py` et de `stats.py` (comme `proportion`, `transcription_complementaire`).

**Chaima TOUNSI OMEZZINE** a proposé les versions primaires des fonctions `total_nucleotide`, `nombre_nucleotide` dans `utility.py`

**Céline ZHANG** a codé plusieurs fonctions statistiques se trouvant dans `stats.py` et `utility.py` (comme les fonctions `taille_ensemble`, `moyenne`, `mediane`, `quartile`, `variance`, etc.)

### Résolution des difficultés rencontrées

Il y a eu des problèmes diverses rencontrés, la résolution s'est fait en équipe. Céline ZHANG avait remarqué un problème d'import des fichiers dans lesquels se trouvait les codes à tester, Omar CHIDA a réglé le problème. Mathis DUMAS a avait quelques soucis sur ses fonctions qu'il a pu debug. Chaima TOUNSI avait des conflicts lors des pull, push, merge, elle a pu réglé ça avec Omar.

### Optimisation des solutions proposées

Dans nos fonctions, les membres ont constaté qu'il pouvait y avoir des améliorations en celui-ci, comme les ajouts de fonctionnalité, faire dans `fasta_to_genome` le changement direct du T en U, les améliorations du dictionnaire, et les optimisations de `transcription_complementaire` en modifiant l'itération. Tous les membres ont participés à la réflexion et proposition de solution.

## Vérification simpliste des tests

Parmis les tests déjà réalisés, nous avons vérifié la validité pour tout le monde, puis nous avons réalisé des tests simples pour les fonctions (`fasta_to_genome`, `transcription_complementaire`, etc.) que nous avons trouvé urgent de tester à ce moment là pour la suite du parcours (bien sûr elles vont être retesté formellement).

## Planification du *sprint*

Enfin, la réunion se termine par la planification de notre premier vrai *sprint*. On sélectionne les tâches à faire pour le cycle (une semaine), et on pré-répartie quelques tâches par préférence ou compétence des membres, et le reste est à prendre dès qu'on finit nos tâches (l'asignement se fait sur Trello).

**Omar CHIDA** doit continuer dans les fonctions basiques, avance sur la documentation des autres parties.

**Mathis DUMAS** doit tester ses fonctions et réfléchir l'affichage des tests d'échantillon.

**Chaima TOUNSI OMEZZINE** doit tester ses fonctions et réfléchir sur la partie des acides aminés.

**Céline ZHANG** doit tester ses fonctions, continuer sur les fonctions de statistiques descriptives, et réfléchir à comment appliquer sur le génome.

## *TO-DO LIST*

- Continuer et finir les fonctions statistiques (interquartile, etc.) de `stats.py`
- Continuer les fonctions basiques de `utility.py` (pour nucléotides, acides, échantillon)
- Faire et écrire les tests des fonctions déjà réalisée (le plus possible)

*Prochaine réunion : 21/11/2020*

## Réunion d'équipe du 21 novembre 2020

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		3h	Discord

### Ordre du jour

1. Avancement de chacun sur le projet (première partie test et deuxième partie)
2. Résolution des difficultés rencontrées
3. Optimisation des solutions proposées
4. Vérification simpliste des tests
5. Planification du prochain *sprint*

### Avancement de chacun sur le projet

Nous avons discuté de l'avancement de chacun sur nos tâches, en partageant ce qui a été réalisé.

**Omar CHIDA** a testé `fasta_to_genome` nettoyé un peu les fichiers inutiles, a corrigé le style d'écriture des fonctions (commentaires, tabulations, saut de ligne, etc.), a résolu les conflicts.

**Mathis DUMAS** a testé sa fonction `transcription_complementaire`, a rendu plus souple le code de certaines fonctions (`fasta_to_genome`, `nombre_nucleotide`, etc.) et le dictionnaire des acides aminés.

**Chaima TOUNSI OMEZZINE** a testé ses fonctions `nombre_nucleotide` et `total_nucleotide` et a apporté des optimisations.

**Céline ZHANG** a testé ses fonctions stastiques de `stats.py` et ses fonctions de `utility.py`, a ajouté les fonctions d'application des fonctions d'analyse statistique sur les échantillons de séquences (`moyenne_nucleotides`, `quartile_nucleotides`, `variance_nucleotides`, `taille_genome`, `moyenne_taille_genome`, etc.) qui permettent de décrire le génome du SARS-COV2.

### Résolution des difficultés rencontrées

Céline a un doute avec la fonction quartile après un débat, nous avons préféré demander à Madame MÉZIÈRES, pour lever le doute sur le calcul de la fonction quartile (car il y avait plusieurs interpolation possible). Plusieurs diverses problèmes (coquilles, structures, etc.) ont été réglé.

### Optimisation des solutions proposées

On a proposé une fonction qui regroupe l'application des fonctions statistiques car l'appel de ces fonctions est assez répétitif. Omar a noté cette idée qui se trouvera dans le prochain *sprint*. On a regroupé les fonctions pour nucléotides et acides aminés en un type de fonction ayant pour terminaison `_elements`.

## Vérification simpliste des tests

Les tests proposés ont bien fonctionné et ont été approuvés.

## Planification du prochain *sprint*

**Omar CHIDA** doit réfléchir à la partie concernant les condons et réaliser la fonction `codons`.

**Mathis DUMAS** doit réfléchir à la partie concernant les condons et implémenter le nécessaire pour les applications de `codons`.

**Chaima TOUNSI OMEZZINE** doit regarder la partie pour les acides aminés, faire les applications.

**Céline ZHANG** résoudre le problème de quartile, corriger la fonction `quartile` et finir les tests des fonctions `quartile`, `intervalle_interquartile`.

## *TO-DO LIST*

- Implémenter la fonction `codons`
- Appliquer les fonctions statistiques sur les acides aminés
- Résoudre le problème de la fonction `quartile` et refaire les tests pour les fonctions `quartile` et `intervalle_interquartile`

*Prochaine réunion : 28/11/2020*

## Réunion d'équipe du 28 novembre 2020

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		3h	Discord

### Ordre du jour

1. Avancement de chacun sur le projet (première partie conclusion et deuxième partie test)
2. Résolution des difficultés rencontrées
3. Optimisation des solutions proposées
4. Vérification simpliste des tests
5. Planification du prochain *sprint*

### Avancement de chacun sur le projet

Nous avons discuté de l'avancement de chacun sur nos tâches, en partageant ce qui a été réalisé, et en interprétant nos résultats pour la première partie.

**Omar CHIDA** a fait la fonction `codons` et les écrit des débuts de tests.

**Mathis DUMAS** a fait la fonction `codons_echantillon` qui permet l'application sur un échantillon.

**Chaima TOUNSI OMEZZINE** a écrit la fonction `nombre_element_echantillon` et les fonctions d'applications des fonctions statiques sur les séquences d'acides aminés. En remarquant la redondance de code, elle a proposé une optimisation pour englober toutes les fonctions.

**Céline ZHANG** a regroupé les résultats d'application des fonctions statistiques sur les échantillons de séquence (sur plusieurs allant de 10 à 10000), puis a corrigé la fonction `quartile` et l'a testé (`intervalle_interquartile` aussi).

### Résolution des difficultés rencontrées

Il y a eu des problèmes d'import des séquences du génome, qui a pu être réglé. On a corrigé certains tests (comme pour `nombre_elements`) qui a eu un problème.

### Optimisation des solutions proposées

Après des longues discussions sur la façon d'optimisation, Omar a réalisé la fonction qui permet l'application de toutes fonctions statistiques à n'importe quel échantillon donné dans n'importe quelle base (nucléotides ou acides aminés).

### Vérification simpliste des tests

Il y a eu de petites corrections sur les tests suite à des modifications. Certains comme les tests de la fonction `nombre_elements` ont été refait.

### Planification du prochain *sprint*

**Omar CHIDA** doit finir les tests pour la fonction `codons`, les fonctions type génératrices et commencer les tests.

**Mathis DUMAS** doit combiner les fonctions statistiques à la fonctions `proportions`, et doit commencer à réaliser des graphes qui vont permettre d'illuster les résultats de la première partie.

**Chaima TOUNSI OMEZZINE** doit tester ses fonctions d'application sur les acides aminés pour la deuxième partie, et regarder la troisième partie pour la distance de Levenshtein.

**Céline ZHANG** doit réfléchir à la rédaction de la conclusion de la première partie avec les résultats, finir les tests inachevés, et regarder la troisième partie pour la distance de Levenshtein.

### ***TO-DO LIST***

- Continuer et finir les fonctions type génératrice
- Réfléchir à la conclusion de la première et deuxième partie
- Commencer à écrire les fonctions qui affichent les graphes des résultats statistiques

*Prochaine réunion : 05/12/2020*

## Réunion d'équipe du 5 décembre 2020

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		2h	Discord

### Ordre du jour

1. Avancement de chacun sur le projet (deuxième partie conclusion et troisième partie discussion)
2. Résolution des difficultés rencontrées
3. Optimisation des solutions proposées
4. Vérification simpliste des tests
5. Planification du prochain *sprint*

### Avancement de chacun sur le projet

Nous avons discuté de l'avancement de chacun sur nos tâches, en partageant ce qui a été réalisé, et nous avons interpréter les résultats des applications de Céline ZHANG, sur les échantillons de séquences de nucléotides. On déduit, sur un échantillon de dix mille séquences entières (du mois de octobre et novembre), l'ARNm du génome SARS-COV2 a une taille d'environ 29813 nucléotides, avec un écart-type de 43 et une intervalle interquartile de 35. Il a environ 8901 nucléotides A, 9580 nucléotides U, 5849 nucléotides G, 5474 nucléotides C, avec respectivement d'écart-type 20 pour A, 20 pour U, 11 pour G, 12 pour C, et d'intervalle interquartile 12 pour A, 17 pour U, 4 pour G, 8 pour C. Des applications sur des échantillons de mille et de deux cents ont été faites, on remarque que les moyennes et médianes ne diffèrent que de très peu (de quelques nucléotides) cependant l'écart-type grandissait avec la taille de l'échantillon, en effet plus la taille de l'échantillon est grande plus on trouve d'écart de valeur. Mais cette augmentation est petite (de l'ordre de 20 nucléotides), ceci est minime par rapport à l'augmentation de la taille de l'échantillon.

**Omar CHIDA** a fixé un problème de codons, a refait les tests de celui-ci, a optimisé les fonctions `call_stat_on_echantillon`, `perform_all_stats`, s'est proposé d'implémenter la distance de Levenshtein.

**Mathis DUMAS** a écrit sa fonction `call_stat_prop` pour la fonction `proportions`, a commencé les graphes pour l'illustration de la première et la deuxième partie, et s'est documenté sur la distance de Levenshtein.

**Chaima TOUNSI OMEZZINE** a commencé les tests d'applications des fonctions statistiques sur les séquences d'acides aminés, a continué les tests des fonctions de type `call_stat_on_echantillon`, et s'est documentée pour la troisième partie, la distance de Levenshtein.

**Céline ZHANG** a appliqué les fonctions statistiques sur des échantillons de séquence de nucléotides, a sauvegardé les résultats pour l'interprétation générale, et s'est documentée pour la troisième partie, la distance de Levenshtein.

## Résolution des difficultés rencontrées

Il y a eu des problèmes des conversions de documents, des questions sur le 'Y' qui peut représenter la nucléotide C ou T, donc il y a eu une séance de débat pour finalement envoyer un mail aux encadrants, mais plusieurs solutions possibles ont été pensées. Ceci bloque un peu la partie représentation graphique et l'interprétation des résultats.

## Optimisation des solutions proposées

Céline ZHANG a fait des tests pour un échantillon de dix mille séquences de nucléotides prise entre les mois d'octobre et novembre, il faut envisager de prendre un échantillon des mois de mars et avril pour comparer les changements éventuels.

## Vérification simpliste des tests

Nous avons vu certaines coquilles dans les descriptions de fonctions qui ont été corrigées. Nous avons vérifier les tests des fonctions `proportions` et `total_elements`.

## Planification du prochain sprint

**Omar CHIDA** doit écrire la fonction qui permet de calculer la distance de Levenshtein entre deux chaînes de caractères.

**Mathis DUMAS** doit commencer l'écriture du rapport avec l'introduction et l'état de l'art, et continuer les graphes.

**Chaima TOUNSI OMEZZINE** doit se documenter et discuter avec Céline ZHANG sur l'implémentation pour la quatrième partie de l'algorithme de Needleman-Wunsch, et réfléchir éventuellement à la conclusion des questions 3, 4.

**Céline ZHANG** doit se documenter et discuter avec Chaima TOUNSI OMEZZINE sur l'implémentation pour la quatrième partie de l'algorithme de Needleman-Wunsch, et commencer éventuellement l'écriture des conclusions des questions 1 et 2.

## TO-DO LIST

- Implémenter la fonction calculant la distance de Levenshtein
- Commencer l'écriture du rapport, introduction et l'état de l'art
- Se documenter et réfléchir à l'implémentation de l'algorithme de Needleman-Wunsch
- Réfléchir sur la conclusion des applications de la parties 1, 2, 3 et 4.
- Envoyer un mail qui permet de lever le doute sur le 'Y', START and STOP, et la notion de *dynamic programming*

*Prochaine réunion : 20/12/2020*

## Réunion d'équipe du 20 décembre 2020

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		3h	Discord

### Ordre du jour

1. Avancement de chacun sur le projet (fin de la deuxième partie et troisième partie test, début quatrième partie)
2. Résolution des difficultés rencontrées
3. Optimisation des solutions proposées
4. Vérification simpliste des tests
5. Planification du prochain *sprint*

### Avancement de chacun sur le projet

Nous avons discuté de l'avancement de chacun sur nos tâches après la session d'examens.

**Omar CHIDA** a implémenté la fonction calculant la distance de Levenshtein `lev` et a proposé une version récursive en programmation dynamique `lev_dp`, et a commencé à implémenter la fonction qui donne l'alignement global optimal de deux séquences.

**Mathis DUMAS** a rédigé un début de l'introduction et de l'état de l'art (les parties concernant la distance de Levenshtein et l'algorithme de Needleman-Wunsch) et a ajouté des tests pour la fonctions proportions.

**Chaima TOUNSI OMEZZINE** s'est documentée et a réfléchi à l'implémentation de l'algorithme de Needleman-Wunsch avec Céline ZHANG, en bordant toutes les manières pour le *filling* (en utilisant une matrice de similarité ou une liste de coût de transformation) et pour le *traceback* (en comporant ou bien en utilisant une matrice qui stock les parcours).

**Céline ZHANG** s'est documentée et a réfléchi à l'implémentation de l'algorithme de Needleman-Wunsch avec Chaima TOUNSI OMEZZINE, en bordant toutes les manières pour le *filling* (en utilisant une matrice de similarité ou une liste de coût de transformation) et pour le *traceback* (en comporant ou bien en utilisant une matrice qui stock les parcours).

### Résolution des difficultés rencontrées

Lors de l'implémentation de l'algorithme de Needleman, il y a eu des doutes sur les situations dans lesquelles plusieurs maxima était possible, Omar CHIDA a choisi de favoriser le *match* puis le *left gap*.

### Optimisation des solutions proposées

Pour l'algorithme de Needleman-Wunsch, Céline ZHANG et Chaima TOUNSI OMEZZINE ont conseillé Omar CHIDA de sauvegarder dans une matrice parallèle le parcours menant à chaque case, ceci évite de refaire les tests pour le traceback.

## Vérification simpliste des tests

Les tests de fonctions Levenshtein ont été vérifiés. Céline ZHANG avait ajouté des tests pour la fonction `taille_ensemble` qui ont été vérifiés.

## Planification du prochain *sprint*

**Omar CHIDA** doit continuer la fonction `needleman` et la terminer au mieux avec les tests.

**Mathis DUMAS** doit continuer la rédaction de l'introduction et l'état de l'art pour le rapport ainsi que les graphes.

**Chaima TOUNSI OMEZZINE** doit appliquer les fonctions calculant la distance de Levenshtein sur des paires de séquences de codons.

**Céline ZHANG** doit faire la mise en page du rapport et organiser le rapport en L<sup>A</sup>T<sub>E</sub>X sur Overleaf, et commencer à rédiger les réponses à la première et deuxième parties.

## *TO-DO LIST*

- Finir d'implémenter l'algorithme de Needleman-Wunsch
- Appliquer les fonctions de distance de Levenshtein sur les séquences de codons
- Continuer l'introduction et l'état de l'art
- Faire la mise en page et organiser le rapport

*Prochaine réunion : 23/12/2020*

## Réunion d'équipe du 23 décembre 2020

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		3h	Discord

### Ordre du jour

1. Avancement de chacun sur le projet (fin de la deuxième partie, troisième partie test, quatrième partie discussion)
2. Résolution des difficultés rencontrées
3. Optimisation des solutions proposées
4. Vérification simpliste des tests
5. Planification du prochain *sprint*

### Avancement de chacun sur le projet

**Omar CHIDA** a continué l'implémentation de l'algorithme de Needleman-Wunsch, la fonction `needleman` pour la question 7 et la fonction `needleman_all` question 9.

**Mathis DUMAS** a testé `needleman` et a trouvé des problèmes (de *out of range*) a ajouté les tests de la fonction `test_call_stat_prop`.

**Chaima TOUNSI OMEZZINE** a appliqué la fonction `lev_itr` (la distance de Levenshtein) sur les paires de séquences et a trouvé un problème *RecursionError* en appliquant `lev_dp` d'où la proposition d'une fonction itérative, a structuré la partie gestion de projet dans le rapport, a fait la fonction `start_to_stop`.

**Céline ZHANG** a ajouté des paires de séquences pour l'application de la fonction `lev_itr` (calculant la distance de Levenshtein), a appliqué sur celles-ci et a ajouté les tests pour les fonctions `call_stats`, `call_stats_taille_genome`, `perform_all_stats`, `perform_all_stats_taille`, etc. et a fait la mise en page du rapport.

### Résolution des difficultés rencontrées

Il y a eu un problème de *out of range* sur les fonctions `needleman`. Le soucis a été réglé par Omar CHIDA.

Il y a eu un problème aussi de *RecursionError* dans l'application de la fonction de Levenshtein. On a réglé le soucis par l'implémentation d'une version Levenshtein itérative.

### Optimisation des solutions proposées

Céline ZHANG a fait une fonction `bank_sequences` (et une recursive), qui permet de prendre dans une banque de séquences de nucléotides un échantillon donné (inférieur ou égal à 10000).

### Vérification simpliste des tests

Les nouveaux tests ajoutés ont été vérifiés.

### Planification du prochain *sprint*

**Omar CHIDA** doit finir les tests des fonctions `needleman`, puis doit commencer les performances des fonctions.

**Mathis DUMAS** doit faire des histogrammes pour les résultats statistiques pour la question 2, doit faire les tests pour les fonctions `needleman`.

**Chaima TOUNSI OMEZZINE** doit corriger la fonction `codons`, faire les tests de celle-ci, réappliquer sur les séquences de codons.

**Céline ZHANG** doit écrire les résultats pour la première partie avec les applications et les graphes, appliquer sur un échantillon de 10000 séquences, finir la banque de séquences.

### *TO-DO LIST*

- Finir les fonctions `needleman` avec les tests
- Faire les graphes pour les premières parties
- Corriger la fonction `codons`
- Appliquer sur des échantillons des mois de mars et d'avril
- Continuer les conclusions des premières parties et la gestion de projet

*Prochaine réunion : 26/12/2020*

## Réunion d'équipe du 26 décembre 2020

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		3h	Discord

### Ordre du jour

1. Avancement de chacun sur le projet (troisième partie conclusion, quatrième partie test)
2. Résolution des difficultés rencontrées
3. Optimisation des solutions proposées
4. Vérification simpliste des tests
5. Planification du prochain *sprint*

### Avancement de chacun sur le projet

**Omar CHIDA** a fini d'écrire la fonction `needleman` et a fait des tests simples, a eu des problèmes de tests avec Biopython qui demande un paramètre `gap` en plus et ne renvoie pas forcément les mêmes solutions.

**Mathis DUMAS** a écrit l'introduction du rapport, a fait quelques graphes pour la première partie.

**Chaima TOUNSI OMEZZINE** a corrigé la fonction `codons`, a rajouté d'autres versions de celle-ci pour l'analyse statistique des séquences de codons et a écrit l'état de l'art pour la partie codons.

**Céline ZHANG** a appliqué les fonctions statistiques sur les échantillons des mois de janvier à avril, a test `needleman`, a trouvé un problème sur le nombre de solution et a fini la rédaction de l'état de l'art pour la partie analyse statistique du génome.

### Résolution des difficultés rencontrées

Le problème de la fonction `needleman` ne provient pas vraiment de la fonction, l'affichage des solutions partielles, et de plus Biopython n'utilise pas toujours le même algorithme pour les alignements, il choisit automatiquement le meilleur algorithme pour la situation. La fonction `codons` a eu un problème d'application a été résolu.

### Optimisation des solutions proposées

La fonction `codons` a été améliorée, les soucis de lecture de triplet indéterminé sont ignorés.

### Vérification simpliste des tests

Tous les tests ajoutés sont passés.

### Planification du prochain *sprint*

**Omar CHIDA** doit commencer la performance et doit régler le soucis de test sur `needleman` avec Biopython et faire l'application pour la question 8.

**Mathis DUMAS** doit continuer la rédaction du rapport sur la distance de Levenshtein et éventuellement sur l'algorithme de Needleman-Wunsch.

**Chaima TOUNSI OMEZZINE** doit finir d'écrire l'état de l'art pour la partie concernant les codons et l'implémentation de l'algorithme de celle-ci.

**Céline ZHANG** doit écrire l'implémentation de l'algorithme pour la partie sur les fonctions d'analyse statistique et éventuellement continuer l'écriture de la gestion de projet.

### ***TO-DO LIST***

- Faire l'application décrite par la question 8 pour `needleman`
- Continuer la rédaction des partie Distance de Levenshtein et Algorithme de Needleman-Wunsch
- Finir l'état de l'art sur la partie codons
- Écrire l'implémentation de l'algorithme pour l'analyse statistique et les codons

*Prochaine réunion : 29/12/2020*

## Réunion d'équipe du 29 décembre 2020

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		3h	Discord

### Ordre du jour

1. Avancement de chacun sur le projet (quatrième partie conclusion, review du rapport)
2. Résolution des difficultés rencontrées
3. Optimisation des solutions proposées
4. Vérification simpliste des tests
5. Planification du prochain *sprint*

### Avancement de chacun sur le projet

**Omar CHIDA** a fait les tests de `biopython`, et a fait l'application pour la question 8, elle affiche en couleur le retraçage du tableau étape par étape avec l'apparition de l'alignement global optimal.

**Mathis DUMAS** a écrit l'état de l'art pour la distance de Levenshtein et l'algorithme de Needleman-Wunsch.

**Chaima TOUNSI OMEZZINE** a fini l'état de l'art concernant la partie codons et a écrit l'implémentation des fonctions codons.

**Céline ZHANG** a fini l'écriture de la partie implémentation des algorithmes pour les fonctions d'analyses statistiques du rapport, a vérifié codons, et a corrigé le rapport (relecture).

### Résolution des difficultés rencontrées

La fonction codons a eu des problèmes de tests avec le module `BioSeq`, les documentations proposent une méthode qui diverge de celle que nous montre `BioSeq`, de ce fait plusieurs fonctions codons existent pour différente utilisation (statistiques, distance de Levenshtein, `BioSeq`). Les fonctions `needleman` ont eu des problèmes de test avec `biopython`, cela a été réglé en envoyant une requête aux contributeurs, nous proposant d'utiliser le module `Aligner` au lieu de `pairwise2`.

### Optimisation des solutions proposées

Céline ZHANG a écrit la fonction `bank_sequences` et une version récursive `bank_sequences_rec` afin de sélectionner des échantillons de séquences pour les tests, il suffit de donner un entier qui sera la taille de l'échantillon, pour que la fonction renvoie un échantillon de séquences sans indétermination<sup>55</sup> de taille voulu issue de la banque de séquences<sup>56</sup>. Omar CHIDA a proposé des fonctions pour les tests de performances de nos fonctions, elles se trouvent dans le fichier `performance.py`.

55. sans problème de code IUPAC (K, N, Y, etc.)

56. un fichier de 20000 séquences sélectionnées sur le site de NCBI : <https://www.ncbi.nlm.nih.gov/labs/virus/vssi/#/>

## Vérification simpliste des tests

Les tests des fonctions `codons` ont été vérifiés, ceux des fonctions `needleman` ont été vérifiés (en dure et en utilisant `biopython`).

## Planification du prochain *sprint*

**Omar CHIDA** doit écrire dans le rapport, l'implémentation des fonctions concernant l'algorithme de Needleman-Wunsch, l'`abstract`, doit faire les tests de performance pour `levenshtein` et `needleman`.

**Mathis DUMAS** doit écrire l'implémentation des fonctions concernant la distance de Levenshtein, doit finir l'état de l'art concernant l'algorithme de Needleman-Wunsch.

**Chaima TOUNSI OMEZZINE** doit réaliser et écrire la partie tests et performances pour les fonctions `codons` et les tests de Levenshtein.

**Céline ZHANG** doit réaliser et écrire la partie tests et performances pour les fonctions statistiques, continuer la partie gestion de projet.

## *TO-DO LIST*

- Faire l'`abstract`
- Faire les tests de performances, et les écrire
- Écrire la partie implémentation pour les fonctions distance de Levenshtein et algorithme de Needleman-Wunsch
- Finir l'état de l'art (algorithme de Needleman-Wunsch)
- Finir la gestion de projet

*Prochaine réunion : 02/01/2021*

## Réunion d'équipe du 2 Janvier 2021

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		3h	Discord

### Ordre du jour

1. Avancement de chacun sur le projet (Fin de la quatrième partie, finalisation rapport, réflexion sur la présentation)
2. Résolution des difficultés rencontrées
3. Optimisation des solutions proposées
4. Vérification simpliste des tests
5. Planification du prochain *sprint*

### Avancement de chacun sur le projet

**Omar CHIDA** a rédigé l'abstract, a écrit les tests de performances, a fait les tests de performances pour les `lev` et les `needleman`.

**Mathis DUMAS** a écrit l'implémentation des fonctions concernant la distance de Levenshtein, a fini l'état de l'art concernant l'algorithme de Needleman-Wunsch.

**Chaima TOUNSI OMEZZINE** a fait les tests de performances pour les fonctions `codons`, et a écrit la partie tests et performances pour celles-ci et les tests de Levenshtein.

**Céline ZHANG** a réalisé les tests de performances pour les fonctions statistiques et a écrit la partie tests et performances de celles-ci.

### Résolution des difficultés rencontrées

En raison de la complexité exponentielle des fonctions `lev_rec` et `needleman_all`, nous avons dû ajuster les tests de performances pour celles-ci. Et éventuellement envisager une solution *multi-threading*.

### Optimisation des solutions proposées

Un script écrit en C a été réalisé pour optimiser les mesures de performances, par l'exécution simultanée des fonctions. L'exécution du traçage de `needleman` a été proposée en réponse à la question 8.

### Vérification simpliste des tests

Les tests des fonctions `codons` et `needleman` ont été revues.

### Planification du prochain *sprint*

**Omar CHIDA** doit réaliser le traçage animé pour `needleman`, doit réaliser les mesures de performances en *multi-thread*, et doit compléter la partie tests et performances.

**Mathis DUMAS** doit finir les graphiques statistiques pour le rapport et doit commencer les graphes pour la présentation.

**Chaima TOUNSI OMEZZINE** doit ajouter des précisions pour expliquer le tableau de bord Trello et doit relire le rapport (introduction, implémentation stats, levenshtein, tests et performances stats, etc.).

**Céline ZHANG** doit rédiger le résumé en français du rapport, faire à nouveau l'introduction en ajoutant le cahier des charges, réaliser le bilan, et doit relire les parties du rapport (état de l'art codons, implémentation codons, `needleman`, tests et performances).

### ***TO-DO LIST***

- Faire une exécution animée du *trace back* de `needleman`
- Réaliser les mesures de performances avec le *multi-threading*
- Finir les graphiques pour les stats et faire les graphiques pour la présentation
- Ajouter des précisions pour l'explication du tableau de bord Trello
- Corriger l'introduction, et ajouter un cahier de charges
- Écrire le bilan et le résumé
- Relire les parties du rapport

*Prochaine réunion : 04/01/2021*

## Réunion d'équipe du 4 Janvier 2021

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		4h	Discord

### Ordre du jour

1. Avancement de chacun sur le projet (Fin du rapport, début Présentation et discussion sur les démonstrations)
2. Optimisation des solutions proposées
3. Vérification simpliste des tests
4. Planification du prochain *sprint*

### Avancement de chacun sur le projet

**Omar CHIDA** a terminé la fonction permettant l'affichage du traçage pour l'alignement global.

**Mathis DUMAS** a terminé les graphiques pour les fonctions statistiques.

**Chaima TOUNSI OMEZZINE** a ajouté des précisions pour expliquer le tableau de bord Trello et a relu le rapport.

**Céline ZHANG** a rédigé le résumé en français du rapport, corrigé l'introduction en ajoutant le cahier des charges, a réalisé le bilan, et a relu les parties du rapport.

### Optimisation des solutions proposées

Tentative d'optimisation des fonctions `needleman` pour un *trace back* plus rapide.

### Vérification simpliste des tests

Check de l'ensemble des tests écrits.

### Planification du prochain *sprint*

L'ensemble de l'équipe doit relire le rapport, signaler et corriger les fautes. Elle devra réfléchir à la conclusion du rapport, puis à la présentation et démonstration à suivre.

### *TO-DO LIST*

- Relecture du rapport
- Réfléchir à la conclusion
- Réfléchir à la présentation
- Réfléchir à la démonstration

*Prochaine réunion : 05/01/2021*

## Réunion d'équipe du 5 Janvier 2021

Membres présents	Membres absents	Durée	Lieu
Mohamed-Omar CHIDA Mathis DUMAS Chaima TOUNSI OMEZZINE Céline ZHANG		2h	Discord

### Ordre du jour

1. Avancement du projet (Relecture du rapport, discuter de présentation et démonstration)
2. Vérification de l'ensemble des tests
3. Organisation de la présentation et de la démonstration à venir

### Avancement du projet

L'ensemble de l'équipe a relu le rapport, a corrigé les erreurs aperçues, et a fait la conclusion du rapport.

### Vérification de l'ensemble des tests

Les tests `pytest` ont été exécutés par tous les membres pour s'assurer du bon fonctionnement de ceux-ci. RAS.

### Organisation de la présentation et de la démonstration à venir

Nous avons continué le début du powerpoint qui avait été réalisé durant la réunion précédente. Nous avons discuté des fonctions (`perform_all_stats`, `codons`, `lev`, `needleman_all`, `nw_verbose`) à exécuter lors de la démonstration pendant la soutenance.

### *TO-DO LIST*

- Finir la présentation
- Avoir la démonstration

*Prochaine réunion : 06/01/2021*

## Table des figures

1	Schéma du processus de transcription . . . . .	6
2	Schéma du processus d'épissage . . . . .	7
3	Schéma du processus de traduction . . . . .	7
4	Schéma du processus de réPLICATION . . . . .	8
5	Le processus de la traduction . . . . .	12
6	Phase d'initialisation de l'algorithme de Wagner et Fischer . . . . .	13
7	REMPLISSAGE de la matrice "dp" . . . . .	14
8	Matrice "dp" complète . . . . .	14
9	Exemple d'un alignement entre les mots "maison" et "long", les barres représentent un match, la croix un mismatch et les tirets un gap. . . . .	15
10	Initialisation de l'algorithme de Needleman-Wunsch pour les mots "maison" et "long", en utilisant la table de coûts adjacente. . . . .	15
11	Matrice "dp" remplie pour les mots "maison" et "long" en utilisant la table de coûts illustrée. . . . .	16
12	Illustration des chemins possibles à emprunter pour le traceback de la matrice "dp" pendant l'alignement des mots "maison" et "long". . . . .	16
13	Les quatre meilleurs alignements pour les mots "maison" et "long" avec comme coûts match=1, mismatch=-3 et gap=-2. . . . .	17
14	Application de la fonction <code>fasta_to_genome</code> . . . . .	20
15	Application de la fonction <code>taille_ensemble</code> . . . . .	20
16	Application de la fonction <code>nombre_element_echantillon</code> . . . . .	20
17	Application de la fonction <code>moyenne</code> . . . . .	21
18	Application de la fonction <code>call_stats</code> . . . . .	21
19	Application de la fonction <code>call_stats.taille_genome</code> . . . . .	21
20	Application de la fonction <code>perform_all_stats</code> sur deux échantillons de 10000 séquences . . . . .	22
21	La répartition des nombres de nucléotides G des séquences d'un échantillon de 1000 . . . . .	22
22	Proportions des nucléotides dans un échantillon de 1000 . . . . .	23
23	Application de la fonction <code>perform_all_stats.taille</code> sur deux échantillons de 10000 séquences . . . . .	23
24	La répartition de la taille des séquences de nucléotides dans un échantillon de 1000 . . . . .	24
25	Application de la fonction <code>codons</code> . . . . .	25
26	Application de la fonction <code>codons_v2</code> . . . . .	25
27	Application de la fonction <code>codons_v3</code> . . . . .	26
28	Échantillon de Janvier à Avril 2020 . . . . .	26
29	Échantillon d'Octobre à Novembre 2020 . . . . .	27
30	Nombres d'acides aminés L des séquences d'un échantillon de 1000 séquences . . . . .	27
31	Proportions des acides aminés dans un échantillon de 1000 séquences . . . . .	28
32	Application de la fonction <code>perform_all_stats.taille</code> sur 1000 séquences d'acides aminés . . . . .	28
33	Application de la fonction <code>perform_all_stats.taille</code> sur 10000 séquences d'acides aminés . . . . .	28
34	Nombre de séquences en fonction de leur taille en acides aminés dans un échantillon de 1000 . . . . .	29
35	Application de la fonction <code>lev</code> sur les séquences des Etats-Unis en décembre 2020 et de Chine en janvier 2020 . . . . .	32
36	Application de la fonction <code>lev</code> sur les séquences des Etats-Unis en décembre 2020 et du Brésil en Mars 2020 . . . . .	32

37	Application de la fonction <code>lev</code> sur les séquences de Belgique et d'Allemagne en décembre 2020 . . . . .	33
38	Application de la fonction <code>lev</code> sur deux séquences différentes de la Chine en mars 2020 . . . . .	33
39	Extrait de l'alignement du génome du SARS-CoV-2 avec l'ARNm du vaccin de Pfizer/BioNTech. (Match = 5, Mismatch = -5, gap = -1) . . . . .	39
40	Extraits de l'alignement du génome du SARS-CoV-2 avec l'ARNm du vaccin de Pfizer/BioNTech. (EMBOSS) . . . . .	39
41	Localisation du gène "spike" dans le génome du coronavirus SARS-CoV-2 . . . . .	40
42	Paramètres utilisés par l'algorithme de Needleman EMBOSS . . . . .	41
43	Test de performance de la fonction <code>moyenne</code> sur des listes de taille 1 à 100 000 . . . . .	45
44	Test de performance de la fonction <code>proportions</code> sur des listes de taille 1 à 100 000 . . . . .	46
45	Test de performance de la fonction <code>mediane</code> sur des listes de taille 1 à 100 000 . . . . .	46
46	Test de performance de la fonction <code>quartile</code> sur des listes de taille 1 à 100 000 . . . . .	47
47	Test de performance de la fonction <code>variance</code> sur des listes de taille 1 à 100000 . . . . .	47
48	Test de performance de la fonction <code>ecart_type</code> sur des listes de taille 1 à 100 000 . . . . .	48
49	Test de performance de la fonction <code>intervalle_interquartile</code> sur des listes de taille 1 à 100 000 . . . . .	48
50	Test de performance des fonctions <code>codons</code> , <code>codons_v2</code> et <code>codons_v3</code> . . . . .	49
51	Test de performance des fonctions <code>lev</code> et <code>lev_dp</code> . . . . .	50
52	Test de performance des fonctions <code>lev</code> , <code>lev_dp</code> et <code>lev_rec</code> . . . . .	51
53	Test de performance des fonctions <code>needleman</code> . . . . .	52
54	Test de performance des fonctions <code>needleman</code> , <code>needleman_all</code> et <code>nw_bio_generic</code> (cette dernière est la fonction de biopython) . . . . .	53
55	Test de performance des fonctions <code>needleman</code> , <code>needleman_all</code> et <code>nw_bio_generic</code> (cette dernière est la fonction de biopython) cas moyen . . . . .	53
56	La matrice SWOT du projet . . . . .	55
57	L'organisation du projet sur Trello . . . . .	56
58	L'organisation du Trello en début de projet . . . . .	66
59	L'organisation du Trello en milieu de projet . . . . .	66
60	L'organisation du Trello en fin de projet . . . . .	67

## Liste des tableaux

1	Le cahier des charges . . . . .	5
2	Table des codons ARN . . . . .	11
3	Table des codes IUPAC . . . . .	12
4	Un exemple de matrice de similarité avec un <code>gap=-10</code> . . . . .	34
5	La répartition des rôles . . . . .	54
6	La méthode SMART . . . . .	54
7	Les réunions de groupe . . . . .	58
8	Le bilan du projet . . . . .	60
9	Le bilan d'équipe . . . . .	61
10	Le temps moyen consacré au projet de chaque membre . . . . .	61

## Références

- [1] Wikipédia. Pandémie de covid-19, 2020. [https://fr.wikipedia.org/wiki/Pand%C3%A9mie\\_de\\_Covid-19](https://fr.wikipedia.org/wiki/Pand%C3%A9mie_de_Covid-19).
- [2] Wikipédia. ADN, 2020. [https://fr.wikipedia.org/wiki/Acide\\_d%C3%A9soxyribonucl%C3%A9ique](https://fr.wikipedia.org/wiki/Acide_d%C3%A9soxyribonucl%C3%A9ique).
- [3] Wikipédia. ARN, 2020. [https://fr.wikipedia.org/wiki/Acide\\_ribonucl%C3%A9ique](https://fr.wikipedia.org/wiki/Acide_ribonucl%C3%A9ique).
- [4] Stated Clearly. What is the rna world hypothesis ?, 2016. <https://www.youtube.com/watch?v=K1xnYFCZ9Yg&feature=youtu.be>.
- [5] Khan Academy. Transcription and mrna processing | biomolecules | mcat | khan academy, 2016. <https://www.youtube.com/watch?v=JQIwwJqF5D0&feature=youtu.be>.
- [6] Wikipédia. ARNm, 2020. [https://fr.wikipedia.org/wiki/Acide\\_ribonucl%C3%A9ique\\_messager](https://fr.wikipedia.org/wiki/Acide_ribonucl%C3%A9ique_messager).
- [7] Wikipédia. Épissage, 2020. <https://fr.wikipedia.org/wiki/%C3%89pissage>.
- [8] Khan Academy. Dna replication and rna transcription and translation | khan academy, 2014. <https://www.youtube.com/watch?v=6gUY5NoX1Lk&feature=youtu.be>.
- [9] Annabac. Schéma traduction, 2020. [https://www.annabac.com/sites/annabac.com/files/dépot/images/PB\\_Bac\\_SVT\\_1re\\_2019\\_9782401052222/05222\\_C02\\_F07\\_01.png](https://www.annabac.com/sites/annabac.com/files/dépot/images/PB_Bac_SVT_1re_2019_9782401052222/05222_C02_F07_01.png).
- [10] Wikipédia. RéPLICATION DES VIRUS, 2020. <http://phage-t4-tpe.e-monsite.com/pages/i-les-virus/la-replication-des-virus.html>.
- [11] Suzanne Clancy. Dna transcription, 2008. <https://www.nature.com/scitable/topicpage/dna-transcription-426/>.
- [12] Antoine Ayache and Julien Hamonier. Cours de statistique descriptive, 2020. [http://math.univ-lille1.fr/~ayache/cours\\_SD.pdf](http://math.univ-lille1.fr/~ayache/cours_SD.pdf).
- [13] Wikipédia. Quartile, 2020. <https://fr.wikipedia.org/wiki/Quartile>.
- [14] Wikipédia. Écart interquartile, 2020. [https://fr.wikipedia.org/wiki/%C3%89cart\\_interquartile](https://fr.wikipedia.org/wiki/%C3%89cart_interquartile).
- [15] Wikipédia. Indicateur de dispersion, 2020. [https://fr.wikipedia.org/wiki/Indicateur\\_deDispersion](https://fr.wikipedia.org/wiki/Indicateur_deDispersion).
- [16] Anna Venancio-Marques. Les acides aminés et la synthèse peptidique, 2013. <https://culturesciences.chimie.ens.fr/thematiques/chimie-organique/methodes-et-outils/les-acides-amines-et-la-synthese-peptidique>.
- [17] Wikipédia. Codon, 2020. <https://fr.wikipedia.org/wiki/Codon>.
- [18] Bernadette Féry. La synthèse des protéines, 2004. <https://slideplayer.fr/slide/1327724/>.
- [19] Wikipédia. Nucléotides, 2020. <https://fr.wikipedia.org/wiki/Nucl%C3%A9otide>.
- [20] Vladimir Likić. The needleman-wunsch algorithm for sequence alignment, 2020. <https://www.cs.sjsu.edu/~aid/cs152/NeedlemanWunsch.pdf>.

- [21] François Rechenmann. Alignement optimal et comparaison de séquences génomiques et protéiques, 2005. <https://interstices.info/alignement-optimal-et-comparaison-de-sequences-genomiques-et-proteiques/>.
- [22] MIT. Sequence alignment and dynamic programming, 2005. [https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-096-algorithms-for-computational-biology-spring-2005/lecture-notes/lecture5\\_newest.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-096-algorithms-for-computational-biology-spring-2005/lecture-notes/lecture5_newest.pdf).
- [23] Wikipédia. Delannoy number, 2020. [https://en.wikipedia.org/wiki/Delannoy\\_number](https://en.wikipedia.org/wiki/Delannoy_number).
- [24] Christer O. Kiselman. Asymptotic properties of the delannoy numbers and similar arrays, 2013. <http://ias.sabanciuniv.edu/documents/kiselmapaper.pdf>.
- [25] Bert Hubert. Reverse engineering the source code of the biontech/pfizer sars-cov-2 vaccine, 2020. <https://berthub.eu/articles/posts/reverse-engineering-source-code-of-the-biontech-pfizer-vaccine/>.
- [26] European Bioinformatics Institute. Emboss pairwise sequence aligner, 2020. [https://www.ebi.ac.uk/Tools/psa/emboss\\_needle/](https://www.ebi.ac.uk/Tools/psa/emboss_needle/).
- [27] National Center for Biotechnology Information. S surface glycoprotein [ severe acute respiratory syndrome coronavirus 2 ], 2020. <https://www.ncbi.nlm.nih.gov/gene/43740568>.