Recherche de motifs et alignements de séquences ADN

Thèmes : algorithmique du texte, recherche de motif, programmation dynamique, suites récurrentes

La *bio-informatique* est un domaine scientifique à l'interface entre les sciences du numérique et la biologie. En particulier, elle accorde une grande importance à l'étude des séquences moléculaires (ADN, ARN, protéines) pour laquelle les méthodes d'algorithmique du texte s'appliquent naturellement.

L' ADN (acide désoxyribonucléique) est une molécule formée d'une double chaîne de *nucléotides*. Les nucléotides se distinguent par leur base azotée qui existe en 4 versions : l'adénine (A), la thymine (T), la guanine (G) et la cytosine (C). Cette molécule sert de support d'information pour le vivant. D'un point de vue informatique, une *séquence ADN* est donc un texte formé de caractères `A`, `T`, `G`, ou `C`.

Une caractéristique du vivant est que l'ADN subit des *mutations*. Il arrive fréquemment qu'un nucléotide soit remplacé par un autre (on parle de *substitution*), qu'un nucléotide soit supprimé (*suppression*) ou qu'un nouveau nucléotide soit inséré (*insertion*) dans une séquence. Les méthodes informatiques doivent donc s'adapter à cette particularité.

Dans ce sujet, on s'intéresse à deux problèmes. Le premier est de *détecter un motif* particulier au sein d'une séquence ADN sachant que le motif a pu subir des mutations. Le second consiste à étudier comment une séquence d'ADN a pu être obtenue par mutations d'une autre séquence. On utilise pour cela le concept d'*alignement* de séquences. Les algorithmes d'alignement sont très importants en biologie : ils permettent de reconstruire des arbres phylogénétiques et de mieux comprendre comment s'est déroulée l'évolution des espèces.

> Les programmes à écrire et les résultats demandés font intervenir une valeur `u0` fixée en début de sujet. Cette valeur peut être différente de celle de vos camarades. Pour vous aider à vérifier vos programmes, les résultats attendus pour la valeur `u0 = 115` sont donnés; sur la fiche réponse vous devrez indiquer les résultats obtenus pour **votre** valeur de `u0`.
> **Attention**, pour éviter les incohérences, lorsque vous changez la valeur `u0` dans le code il faut ré-évaluer l'ensemble de votre fichier depuis le départ.

1 Génération de séquences aléatoires

Cette première partie est consacrée à la génération de séquences d'ADN aléatoires. Elles ne sont pas réalistes d'un point de vue biologique mais seront pratiques pour tester les fonctions écrites dans les parties suivantes. Cette génération se base sur votre valeur `u0`.

On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par récurrence :

$$\begin{cases} u_0 & = \texttt{u0} \\ u_{n+1} & = 15091 \times u_n \pmod{64007} \end{cases}$$

Question 1

Construire un tableau Numpy `u` de longueur 100000 où chaque case `u[i]` contient la valeur de u_i. Donner les valeurs de

a. u_{10},
b. u_{100},
c. u_{1000}.

Pour éviter de les recalculer, on se servira de ce tableau dans la suite à chaque fois que l'on a besoin d'une valeur u_i.

Soit $k \in \mathbb{N}$ et $l \in \mathbb{N}$ deux entiers naturels. On définit la séquence aléatoire `S(k, l)` comme une chaîne de caractères de longueur l telle que :

$$\forall i \in [|0, l-1|], \quad \texttt{S(k,l)}[i] = \begin{cases} \texttt{"A"} \text{ si } u_{i+k} = 0 \pmod 4 \\ \texttt{"T"} \text{ si } u_{i+k} = 1 \pmod 4 \\ \texttt{"G"} \text{ si } u_{i+k} = 2 \pmod 4 \\ \texttt{"C"} \text{ si } u_{i+k} = 3 \pmod 4 \end{cases}$$

On remarquera que nous avons suivi la convention de Python qui est d'indicer les chaînes de caractères à partir de 0 (comme pour les listes et les tableaux).

Question 2

Écrire une fonction d'en-tête `S(k, l)` prenant en arguments les entiers k et l et retournant la chaîne de caractères construite par cette méthode. Donner les valeurs de

a. `S(0, 5)`,
b. `S(10, 5)`,
c. `S(20, 5)`.

2 Recherche d'un motif dans une séquence ADN

On procède en deux temps. On écrit d'abord un algorithme permettant de trouver les occurrences exactes d'un motif. Dans un second temps on adapte cet algorithme pour prendre en compte les possibilités de substitutions.

2.1 Recherche exacte

Soit S une séquence de longueur n et M un motif de longueur m. On dit qu'il y a une occurrence du motif M dans S en position i si $S[i]S[i+1]\ldots S[i+l-1] = M$. Par exemple, le motif `AGAGA` possède plusieurs occurrences exactes dans la séquence suivante :

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
A  T  T  A  G  A  G  A  C  T  G  G  T  A  G  A  G  A  G  A  C
         A  G  A  G  A
                                       A  G  A  G  A
                                          A  G  A  G  A
```

en positions 3, 13 et 15. On remarque à l'aide de cet exemple que les occurrences de motifs peuvent se chevaucher dans la séquence.

Question 3

Écrire une fonction `recherche_exacte(seq : str, motif : str) -> list` prenant en arguments une séquence `seq` et un `motif` et retournant la liste des positions où le motif apparaît exactement dans la séquence.

a. Donner la liste des positions où le motif `TATA` apparaît dans `ATATATACATATA`,
b. puis dans `S(20, 1000)`.
c. Combien de fois le motif `TATA` apparaît-il dans `S(50, 10000)` ?

INDICATIONS

Utiliser l'algorithme de la *fenêtre glissante* : considérer toutes les positions `i` où le motif peut apparaître et pour chaque valeur de `i` considérée, tester lettre par lettre s'il y a une occurrence du motif à cette position. Pour chaque occurrence trouvée on a ajoute sa position dans une liste résultat qu'il suffira de retourner en fin de fonction.

2.2 Recherche avec substitutions

On veut maintenant pouvoir détecter la présence d'un motif dans une séquence ADN en prenant en considération que le motif a possiblement subi des mutations. On ne s'intéressera qu'au cas des substitutions.

Pour des raisons bio-chimiques, les substitutions n'ont pas toute la même probabilité de survenir. Par exemple, il est bien plus fréquent qu'un `A` soit remplacé par un `G` que par un `T`. Pour quantifier ces différentes probabilités, on introduit généralement une *matrice de coût de substitution*. Nous utiliserons celle-ci :

	A	**G**	**C**	**T**
A	0	2	6	8
G	2	0	9	6
C	6	9	0	1
T	8	6	1	0

Ainsi d'après cette matrice, remplacer un `A` par un `G` coûte 2 tandis que remplacer un `G` par un `C` coûte 9. Remplacer une lettre par elle même n'est pas vraiment une mutation donc le coût est nul dans ce cas.

La matrice de substitution sera codée en Python à l'aide d'un dictionnaire que l'on pourra considérer ici comme un tableau indexé par des couples de lettres. On donne le début du code à écrire pour construire la matrice :

```
matsub = {}
matsub['A', 'A'] = 0
matsub['A', 'G'] = 2
...
```

Soit S une séquence de longueur n et M un motif de longueur m. On dira que le motif M apparaît dans S en position i avec un coût C lorsque

$$\sum_{k=0}^{m-1} \texttt{matsub}[S[i+k],\, M[k]] = C.$$

Par exemple, si on s'interesse au motif `TACT` dans

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
A  T  T  A  C  T  G  A  C  T  G  G  T  A  G  A  G  A  G  A  C
      T  A  C  T
                           T  A  C  T
                                    T  A  C  T
```

on peut dire qu'il apparaît en position 2 avec un coût de 0 (occurrence exacte), en position 9 avec un coût de $2 + 9 = 11$ ($G \to A + G \to C$) et en position 12 avec un coût de $9 + 8 = 17$ ($G \to C + A \to T$).

En réalité, chaque position dans la séquence correspond à un coût d'apparition du motif et il conviendra donc de se fixer le seuil que l'on est prêt à tolérer.

Question 4

Écrire une fonction `recherche_subs(seq : str, motif : str, coutmax : int) -> list` retournant la liste des positions dans la séquence `seq` où le `motif` apparaît avec un coût inférieur ou égal à `coutmax`.
On se fixe `coutmax = 3`.

a. Donner la liste des positions où le motif `TATA` est détecté dans `ATATATACATATA`.
b. Donner les positions des 5 premières apparitions de `TATA` dans $S(50, 10000)$.
c. Combien de fois détecte-t-on le motif `GATTACA` dans $S(50, 10000)$?

INDICATIONS

Reprendre l'algorithme de la *fenêtre glissante* de la question précédente. Cette fois-ci pour chaque position `i` considérée, calculer le coût de l'occurrence à cette position. Si le coût est inférieur à `coutmax` alors sauvegarder dans une liste résultat la position `i`.