

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 u_0 fixée en début de sujet. Cette valeur sera très probablement différente de celle de vos camarades. Pour vous aider à vérifier vos programmes, les résultats attendus pour la valeur $u_0 = 115$ sont donnés; sur la fiche réponse vous devrez indiquer les résultats obtenus pour votre valeur de u_0 .

Attention, pour éviter les incohérences, lorsque vous changez la valeur u_0 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 u_0 .

On considère la suite $(u_n)_{n \in \mathbb{N}}$ définie de la façon suivante :

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

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

- u_{10} ,
- u_{100} ,
- 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 .

Soient $k \in \mathbb{N}$ et $l \in \mathbb{N}$ deux entiers positifs. On définit la séquence pseudo-aléatoire $S(k, l)$ comme une chaîne de caractères de longueur l telle que

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

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 de signature $S(k: \text{int}, l: \text{int}) \rightarrow \text{str}$ prenant en arguments les entiers k et l et renvoyant la chaîne de caractères construite par cette méthode. Donner les valeurs de

- $S(0, 5)$,
- $S(10, 5)$,
- $S(20, 5)$.

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

Soient S une séquence de longueur n et M un motif de longueur m . On dit qu'il y a une **occurrence exacte** du motif M dans S en position i si $S[i]S[i+1]\dots S[i+m-1] = M$. 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 exactes 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 renvoyant la liste des positions des occurrences exactes du motif dans la séquence.

- Donner la liste des positions des occurrences exactes du motif TATA dans ATATATACATATA,
- puis dans $S(20, 1000)$.
- Combien y a-t-il d'occurrences exactes du motif TATA 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 exacte du motif à cette position. Pour chaque occurrence exacte trouvée on ajoute sa position dans une liste résultat qu'il suffira de renvoyer 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 biochimiques, 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	T	G	C
A	0	8	2	6
T	8	0	6	1
G	2	6	0	9
C	6	1	9	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
...
```

Soient S une séquence de longueur n et M un motif de longueur m . On dira qu'il y a une **quasi-occurrence** du motif M dans S en position i avec un coût C lorsque

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

Par exemple, si on s'intéresse 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 observe des quasi-occurrences en position 02 avec un coût de 0 (c'est donc en fait une occurrence exacte), en position 09 avec un coût de $2 + 9 = 11$ ($(G \rightarrow A) + (G \rightarrow C)$), et en position 12 avec un coût de $9 + 8 = 17$ ($(G \rightarrow C) + (A \rightarrow T)$).

En réalité, chaque position dans la séquence correspond à une quasi-occurrence pour un certain coût d'apparition du motif, et il conviendra donc de choisir une borne supérieure pour les coûts que l'on est prêt à tolérer.

Question 4. Écrire une fonction `recherche_subs(seq: str, motif: str, cmax: int) -> list` renvoyant la liste des positions des quasi-occurrences de `motif` dans `seq` ayant un coût inférieur ou égal à `cmax`.

Pour les calculs ci-dessous, on se fixe `cmax = 3`.

- Donner la liste des positions des quasi-occurrences du motif TATA dans ATATATACATATA.
- Donner les positions des 5 premières quasi-occurrences de TATA dans `S(50, 10000)`.
- Combien y a-t-il de quasi-occurrences du 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 la quasi-occurrence à cette position. Si le coût est inférieur à `cmax` alors sauvegarder dans une liste résultat la position `i`.

3 Alignement de séquences ADN (Needleman-Wunsch 1970)

Dans cette partie on s'intéresse à l'alignement de deux séquences d'ADN. Considérons par exemple les séquences GAATTCAGTTA et AAGTCAGTTA. Une étude attentive des deux séquences permet de constater qu'elles sont similaires, il est donc possible que la seconde séquence soit en fait le résultat de plusieurs mutations sur la première séquence.

Aligner les deux séquences consiste à insérer des symboles - (appelés *gap*) dans chacune des séquences. Les gaps représentent les *suppressions* et les *insertions*, et sont disposés de manière à ce que les séquences se correspondent le mieux possible. Par exemple, on peut considérer l'alignement suivant :

```

00 01 02 03 04 05 06 07 08 09 10 11
G  A  A  T  T  C  A  G  -  T  T  A
A  A  G  -  T  C  A  G  T  T  T  A

```

Cet alignement traduit le fait que la seconde séquence peut s'obtenir à partir de la première avec 2 substitutions (en positions 0 et 2), 1 suppression d'un T (en position 3) et 1 insertion d'un T (inséré en 8^e position).

L'algorithme de Needleman-Wunsch permet de calculer un alignement optimal de deux séquences, c'est-à-dire qui correspond au plus faible coût en mutations. Il repose sur la *programmation dynamique*.

3.1 Coût d'un alignement

Pour donner un *coût* à un alignement, on utilisera la matrice de substitution de la partie précédente et on se fixera arbitrairement un coût de suppression $c_s = 5$ et un coût d'insertion $c_i = 4$.

Soit (s_1, s_2) un couple de séquences ; un *alignement* est un couple (r_1, r_2) où r_1 (resp. r_2) a été obtenu à partir de s_1 (resp. s_2) par insertions de symboles de gaps et tel que pour tout i , on n'a pas $r_1[i] = r_2[i] = -$ (pas d'alignement de symboles gaps).

Le *coût de l'alignement* (r_1, r_2) est $\sum_{k=0}^{n-1} \alpha_k$ avec $n = |r_1| = |r_2|$ et

$$\alpha_k = \begin{cases} c_i & \text{si } r_1[k] = - \text{ (insertion)} \\ c_s & \text{si } r_2[k] = - \text{ (suppression)} \\ \text{mat}[r_1[k], r_2[k]] & \text{sinon (correspondance ou substitution).} \end{cases}$$

Question 5. Écrire une fonction `coût_alignement(r1 : str, r2 : str) -> int` qui prend en arguments un alignement supposé valide et qui renvoie son coût. (Facultatif: vérifier la validité des entrées à l'aide d'assertions.) Donner le coût des alignements suivants :

- ("AATTGCAT", "AA--GCAT")
- ("A-TTGCA-", "AAT--CAT")
- `S(2000, 50)`, `S(3000, 50)`

3.2 Coût d'alignement optimal

Un alignement de deux séquences (s_1, s_2) est *optimal* si son coût est minimal parmi l'ensemble des alignements possibles de ces séquences.

On remarque que lorsqu'on forme un alignement optimal, une et une seule des situations suivantes se produit :

- la dernière lettre X de s_1 est alignée avec la dernière lettre Y de s_2 ;
- la dernière lettre de s_1 est supprimée ;
- la dernière lettre de s_2 est une insertion.

Dans chacun de ces cas, le reste de l'alignement, c'est-à-dire r_1 et r_2 privés de leur dernier symbole, est aussi un alignement optimal (sinon l'alignement de départ ne peut pas l'être).

Ainsi un alignement optimal est formé à partir d'un alignement optimal d'un sous-problème, la programmation dynamique est donc adaptée.

Soient s_1 et s_2 deux séquences de longueur respective n et m . Pour $(i, j) \in \llbracket 0, n \rrbracket^2$, on note $T[i, j]$ le coût d'un alignement optimal entre les i premières lettres de s_1 (préfixe de s_1 de longueur i) et les j premières lettres de s_2 (préfixe de s_2 de longueur j).

Question 6. Parmi les schémas de programmation dynamique suivants, indiquer celui qui correspond à la méthode proposée :

- $\forall i > 0, \forall j > 0, \quad T[i, j] = \min(T[i-1, j-1] + \text{matsub}[s_1[i-1], s_2[j-1]],$
 $\quad T[i-1, j] + c_i,$
 $\quad T[i, j-1] + c_s)$
- $\forall i > 0, \forall j > 0, \quad T[i, j] = \min(T[i-1, j-1] + \text{matsub}[s_1[i-1], s_2[j-1]],$
 $\quad T[i-1, j-1] + c_s,$
 $\quad T[i-1, j-1] + c_i)$
- $\forall i > 0, \forall j > 0, \quad T[i, j] = \min(T[i-1, j-1] + \text{matsub}[s_1[i-1], s_2[j-1]],$
 $\quad T[i-1, j] + c_s,$
 $\quad T[i, j-1] + c_i)$
- $\forall i > 0, \forall j > 0, \quad T[i, j] = \min(T[i-1, j-1] + \text{matsub}[s_1[i-1], s_2[j-1]],$
 $\quad T[i-1, j-1] + c_s,$
 $\quad T[i-1, j-1] + c_i)$

On remarquera que les conditions initiales sont données par

$$\begin{cases} T[0, j] = j \times c_i \\ T[i, 0] = i \times c_s \end{cases}$$

car dans le premier cas, on n'a pas d'autre choix que d'insérer toutes les lettres de s_2 et dans le second cas, il faut nécessairement supprimer toutes les lettres de s_1 .

Question 7. Écrire une fonction `coût_optimal(s1 : str, s2 : str) -> int` prenant en entrée une paire de séquences et renvoyant le coût d'un alignement optimal. Donner le coût d'alignement optimal des séquences suivantes.

- ("GGTTCA", "GGTCA")
- ("CATTACATCTTTAGCA", "TTTCAGATCTCTATGCA")
- (S(2000, 50), S(3000, 50))

Indications. On met en œuvre la programmation dynamique: on construit une matrice T de dimensions $(n + 1) \times (m + 1)$ et on la remplit à l'aide des conditions initiales données et de la relation de récurrence trouvée à la question 6. La valeur que l'on cherche est $T[n, m]$.

3.3 Reconstruction d'un alignement optimal

Question 8. Écrire une fonction `alignement_optimal(s1 : str, s2 : str) -> (str, str)` prenant en arguments une paire de séquences et renvoyant un alignement optimal de ces deux séquences.

Donner des alignements optimaux de :

- a) ("GGTTCA", "GGTCA")
- b) ("CATTACATCTTTAGCA", "TTTCAGATCTCTATGCA")
- c) (S(2000, 50), S(3000, 50))

Indications. Il s'agit de reconstruire la solution obtenue par programmation dynamique à la question précédente. Pour cela, on pourra reprendre le code de la question 7 et ajouter une matrice U de mêmes dimensions que T qui permet de sauvegarder le choix effectué à chaque étape :

- $U[i, j] = 1$ si $T[i, j]$ a été obtenu par substitution (cas 1),
- $U[i, j] = 2$ si $T[i, j]$ a été obtenu par suppression (cas 2),
- $U[i, j] = 3$ si $T[i, j]$ a été obtenu par insertion (cas 3).

Une fois les matrices T et U construites par programmation dynamique, on reforme l'alignement optimal à l'aide de la matrice U , en partant de la case $U[n, m]$ et en remontant jusqu'au cas initial. L'alignement est ainsi reconstruit de la droite vers la gauche (des derniers symboles aux premiers symboles).