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.

Le TP sera entièrement réalisé en langage C.

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 :

$$\begin{cases} u_0 & = 115 \\ u_{n+1} & = 15091 \times u_n \pmod{64007} \end{cases}$$

Question 1

Déclarer un tableau global `u` de longueur 100000; puis dans la fonction `main` remplir chaque case `u[i]` avec la valeur 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 `void S(char t[], int k, int l)` prenant en arguments les entiers k et l et calculant la chaîne de caractères construite par cette méthode. Cette chaîne sera enregistrée dans `t` supposé assez grand. 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
`int recherche_exacte(char seq[], int n, char motif[], int m, int res[])`
prenant en arguments une séquence `seq` de longueur `n` et un `motif` de longueur `m` et retournant la liste des positions où le motif apparaît exactement dans la séquence. Cette liste sera enregistrée dans `res` supposé assez grand et la fonction retourne le nombre d'occurrences.

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

On utilisera l'algorithme naïf 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 subsitution sera codée en C par la variable :

```c
int matsub[4][4];
```

et on pourra se servir de la fonction suivante :

```c
// Retourne le numéro du caractère A, G, C ou T dans cet ordre
int numlettre(char c) {
    switch (c) {
        case 'A':
            return 0;
        ...
        default:
            exit(1);
    }
}
```

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

```
int recherche_subs(char seq[], int n, char motif[], int m,
                   int res[], int coutmax)
```

fonctionnant sur le même principe que `recherche_exacte` mais retournant la liste des occurrences ayant un coût inférieur 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`.

3 Alignement de deux 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 `AAGTCAGTTTA`. 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 subsitutions (en positions 0 et 2), 1 suppression d'un `T` (en position 3) et 1 insertion d'un T (inséré en 8e 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

$$\alpha_k = \begin{cases} c_i \text{ si } r_1[k] =' -' \text{ (insertion)} \\ c_s \text{ si } r_2[k] =' -' \text{ (suppresion)} \\ \texttt{matsub}(r_1[k], r_2[k]) \text{ sinon (correspondance ou substitution)} \end{cases}$$

Question 5

Écrire une fonction
`int cout_alignement(char r1[],char  r2[], int n)`
qui prend en arguments un alignement supposé valide de longueur n et qui retourne son coût.
Donner le coût des alignements suivants :
a. `("AATTGCAT", "AA--GCAT")`
b. `("A-TTGCA-", "AAT--CAT")`
c. `(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 :

1. la dernière lettre X de s_1 est alignée avec la dernière lettre Y de s_2,
2. la dernière lettre de s_1 est supprimée,
3. 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 l'est pas).

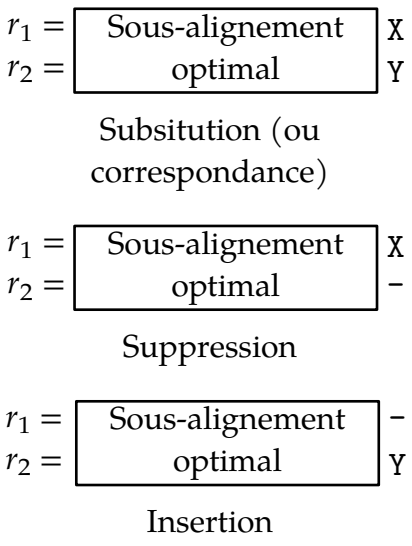

Figure 1 Alignement optimal : différents cas possibles

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

Soit s_1 et s_2 deux séquences de longueurs n et m. Pour $0 \leq i \leq n$ et $0 \leq j \leq n$, on note $T[i, j]$ le coût d'un alignement optimal entre les i premières lettres de s_1 (préfixe 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 :

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

$$B) \quad \forall i > 0,\ \forall j > 0, \quad T[i,j] = \max \begin{cases} T[i-1, j-1] + matsub[s_1[i-1], s_2[j-1]] \\ T[i-1, j-1] + c_s \\ T[i-1, j-1] + c_i \end{cases}$$

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

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

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
`int cout_optimal(char s1[], int n, char s2[], int m)`
prenant en entrée une paire de séquences et retournant le coût d'un alignement optimal.
Donner le coût d'alignement optimal des séquence suivantes.
a. `("GGTTCA", "GGTCA")`
b. `("CATTCACATCTTTAGCA", "TTTCAGATCTCTATGCA")`
c. `(S(2000, 50), S(3000, 50))`

INDICATIONS

On met en œuvre la programmation dynamique : on construit une matrice `T` de dimensions $LMAX \times LMAX$ avec par exemple $LMAX = 300$ et on la remplit sur $(n+1) \times (m+1)$ à l'aide des conditions initiales données et de la relation de réccurrence 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
```
int alignement_optimal(char s1[], int n, char s2[], int m,
                       char r1[], char r2[])
```
prenant en arguments une paire de séquence et retournant un alignement optimal de ces deux séquences. Cet alignement sera enregistré dans `r1` et `r2` supposés assez grands et on retournera la longueur de l'alignement.
Donner des alignements optimaux de :
a. `("GGTTCA", "GGTCA")`
b. `("CATTCACATCTTTAGCA", "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).