TD : HACHAGE POLYNOMIAL – MOTIF ADN

(CCINP-TSI-Info 2020)

Le thème traité est la recherche d'un motif dans une molécule d'ADN.

Une molécule d'ADN est constituée de deux brins complémentaires, qui sont un long enchaînement de nucléotides de quatre types différents désignés par les lettres A, T, C et G. Les deux brins sont complémentaires : « en face » d'un 'A', il y a toujours un 'T', et « en face » d'un 'C', il y a toujours un 'G'. Pour simplifier le sujet, une molécule d'ADN est considérée comme une chaîne de caractères sur l'alphabet {A, C, G, T} (on s'intéresse donc seulement à l'un des deux brins). On parlera de séquence d'ADN.

Ce TD reprend la dernière partie du sujet. Dans les parties précédentes, différentes approches ont été étudiées pour rechercher un motif dans une séquence d'ADN. Après avoir modélisé une molécule d'ADN comme une chaîne de caractères sur l'alphabet {A, C, G, T}, un algorithme naïf de recherche a été étudié. Il consistait à rechercher un motif M à chaque position possible. Cependant, la complexité en O(nm), où n est la taille de la séquence et m la taille du motif, devient rapidement prohibitive pour des séquences de taille réaliste.

Ensuite, l'algorithme de Knuth–Morris–Pratt a été introduit pour améliorer la recherche de motifs. Il permet de réduire le nombre de comparaisons en évitant les répétitions inutiles grâce à une analyse des préfixes (début de M, sans prendre M en entier) et des suffixes (fin de M, là encore sans prendre M en entier) du motif. De cette manière, la structure du motif est mieux exploitée et le temps d'exécution est amélioré.

Une autre approche consistait à construire une liste triée de sous-mots pour pouvoir utiliser une recherche dichotomique. Au lieu de parcourir directement la chaîne S, on construit d'abord une structure de données (une liste triée de sous-mots), puis on effectue une recherche efficace dans cette liste. Cette méthode sépare deux opérations : le pré-traitement coûteux (générer tous les sous-mots puis les trier) et la recherche rapide ensuite (recherche dichotomique).

La partie que nous allons étudier introduit une nouvelle méthode, fondée sur l'utilisation d'une fonction de hachage, comme dans l'algorithme de Karp–Rabin. L'idée principale est d'associer à chaque motif une valeur numérique, obtenue en interprétant la suite de caractères comme l'écriture d'un entier en base 4, puis en évaluant un polynôme. On compare alors les valeurs de hachage plutôt que les motifs eux-mêmes, ce qui permet d'accélérer significativement la recherche, surtout si cette évaluation est optimisée grâce à l'algorithme de Horner. Cette approche complète les méthodes précédentes et illustre le lien entre algorithmes de recherche, complexité et outils mathématiques.

I) FONCTION DE HACHAGE, ALGORITHME DE KARP-RABIN

Certains algorithmes, comme l'algorithme de Karp-Rabin (1987), utilisent une fonction de hachage h qui à un motif renvoie une valeur numérique.

Voici un exemple de fonction de hachage :

- à chaque caractère de l'alphabet, on associe une valeur. Ici, on va associer à 'A' la valeur 0, à 'C' la valeur 1, à 'G' la valeur 2 et à 'T' la valeur 3. Pour un motif de taille n, on obtient donc une suite de chiffre a_{n-1} … a_1, a_0. Par exemple, à la chaîne 'TAGC', on lui associe la suite de chiffre 3021 ;
- cette suite de chiffre est considérée comme l'écriture d'un entier en base b, où b est le nombre de caractères présents dans l'alphabet. On a donc ici b = 4;
- on calcule ensuite cet entier en base 10 (on calcule donc $a_{n-1}b^{n-1} + … + a_1b^1 + a_0b^0$) ;
- puis on calcule le reste de la division euclidienne de ce nombre par 13.

1. On ne considère que des motifs de taille 3. Que renvoie la fonction de hachage avec les motifs 'CCC', 'ACG', 'GAG' ? On détaillera les calculs.

2. Écrire la fonction `f_CodeBase4(sequence)` qui prend en entrée une chaine de 3 caractères (comme 'CCC', 'ACG', 'GAG', …) et renvoie la valeur sous forme d'une liste en base 4.

```
Tester :        >>> f_CodeBase4('AAT')        >>> f_CodeBase4('GTT')
                [0, 0, 3]                     [2, 3, 3]
```

3. Écrire la fonction `f_HashCode(val_b4)` qui prend en entrée la liste de la valeur en base 4 et renvoie la valeur du hash code en suivant les opérations données dans l'exemple.

```
Tester :        >>> f_HashCode([1,1,1])        >>> f_HashCode([2,0,2])
                8                             8
```

Dans cette fonction de hachage, nous avons besoin de transformer un entier en base b en un entier en base 10. On remarque que l'on peut éventuellement faire ce calcul en évaluant un polynôme.

II) ÉVALUATION DE POLYNÔME, ALGORITHME DE HÖRNER

Dans cette sous-partie, nous allons nous intéresser à l'évaluation d'un polynôme et de son coût lorsque l'on compte les multiplications, les additions et les affectations comme des opérations unitaires.

Soit $P = \sum_{k=0}^{n} a_k X^k$ un polynôme, il sera représenté par la liste $[a_n, … ,a_0]$.

4. Écrire une fonction `f_Eval(P, b)` ayant pour paramètre un polynôme P (donc une liste de nombres $[a_n, … , a_0]$) et un nombre b. Cette fonction doit renvoyer la valeur de P en b.

```
Tester :        >>> f_Eval([2,0,2],4)         >>> f_Eval([0,1,2],4)
                34                            6
```

5. Combien faut-il de multiplications pour calculer un b^k ? Combien de fois doit-on le faire ? Combien faut-il de multiplications pour calculer chaque $a_k \cdot b^k$? Combien de fois doit-on le faire ? Combien faut-il d'addition pour calculer la somme totale ? En déduire la complexité de la fonction `eval()`.

On peut être plus astucieux en utilisant l'algorithme de Hörner qui se base sur l'égalité suivante :

$$P(X) = \Big(\big(\ldots\big((a_n X + a_{n-1})X + a_{n-2}\big)X + \ldots\big)X + a_1\Big)X + a_0$$

Plus précisément, pour évaluer P(b), on commence par calculer $a_n \cdot b + a_{n-1}$ puis on multiplie le résultat par b et on ajoute a_{n-2}, etc.

6. Quelle est la complexité avec cette méthode ?

7. Écrire une fonction itérative `f_Hornerit(P, b)` ayant pour paramètres un polynôme P, sous forme de liste, ainsi qu'un réel b, et renvoyant P(b) en utilisant l'algorithme de Hörner.

Tester :

```
>>> f_Hornerit([2,0,2],4)
34
```

```
>>> f_Hornerit([0,1,2],4)
6
```

8. Compléter la fonction `f_Hornerrec()` pour avoir une fonction récursive qui évalue un polynôme en utilisant l'algorithme de Homer.

```
def f_Hornerrec(P,b):
if ……….
        return ……..
    else :
        s = P[len(P) - 1]
        s1 = P[0:len(P) - 1]
        return ……..
```

Tester :

```
>>> f_Hornerrec([2,0,2],4)
34
```

```
>>> f_Hornerrec([0,1,2],4)
6
```