

Rapport du Projet LU3IN003

ZHOU runlin 28717281

MA peiran 28717249

Introduction :

Le but de ce projet porte est un problème de génomique : l'**alignement de séquences**.

D'un point de vue biologique, il s'agit de mesurer la similarité entre deux séquences d'ADN, que l'on voit simplement comme des suites de nucléotides.

D'un point de vue informatique, les séquences de nucléotides sont vues comme des mots sur l'**alphabet** $\{A, T, G, C\}$. On s'intéresse d'abord à un *algorithme naïf*, puis à un algorithme de *programmation dynamique*. On utilise la méthode diviser pour régner pour améliorer la complexité spatiale de ces algorithmes.

Nous présenterons notre réalisation de l'algorithme en plusieurs parties.

Le problème d'alignement de séquences (Ex 2) :

Question 1:

Soit $(\overline{xu}, \overline{yv})$ est un alignement de (xu, yv) ssi

$$\left| \begin{array}{l} \text{(i) } \pi(\overline{xu}) = xu \\ \text{(ii) } \pi(\overline{yv}) = yv \\ \text{(iii) } |\overline{xu}| = |\overline{yv}| \\ \text{(iv) } \forall i \in [1, \dots, |\overline{xu}|], \overline{xu}_i \neq - \text{ ou } \overline{yv}_i \neq - \end{array} \right.$$

Nous pouvons facilement déduire que :

- $\pi(\overline{xu}) = \pi(\overline{x}) * \pi(\overline{u}) = xu$
- $\pi(\overline{yv}) = \pi(\overline{y}) * \pi(\overline{v}) = yv$
- $|\overline{xu}| = |\overline{x}| + |\overline{u}|$, car $(\overline{x}, \overline{y})$ et $(\overline{u}, \overline{v})$ sont respectivement des alignements de (x, y) et (u, v) , donc on a $|\overline{x}| + |\overline{u}| = |\overline{y}| + |\overline{v}| = |\overline{yv}|$
- Car $(\overline{x}, \overline{y})$ et $(\overline{u}, \overline{v})$ sont respectivement des alignements de (x, y) et (u, v) , donc on a $\forall i \in [1, \dots, |\overline{x}|], \overline{x}_i \neq - \text{ ou } \overline{y}_i \neq -$ et $\forall i \in [1, \dots, |\overline{u}|], \overline{u}_i \neq - \text{ ou } \overline{v}_i \neq -$, on ajoute les deux conditions ensemble. Et on a $\forall i \in [1, \dots, |\overline{xu}|], \overline{xu}_i \neq - \text{ ou } \overline{yv}_i \neq -$

Donc, $(\overline{xu}, \overline{yv})$ est bien que un alignement de (xu, yv)

Question 2 :

Si $(\overline{x}, \overline{y})$ est un alignement de (x, y) , on a $\forall i \in [1, \dots, |\overline{x}|], \overline{x}_i \neq - \text{ ou } \overline{y}_i \neq -$
donc la longueur maximale est $|x| + |y|$

Algorithmes pour l'alignement de séquences (Ex 3) :

Programmation naïve :

Question 3 :

Nous pouvons convertir le problème en sélectionnant k dans $n + k$ positions comme '-', donc le nombre de mot est C_{n+k}^n

Question 4 :

Soit $k \in [0, m]$ (l'intervalle étant choisi de manière à ne pas dépasser le nombre maximum de gaps, déterminé à la question (2)).

On suppose que l'on ajoute k gaps à x pour former une chaîne \bar{x} de longueur $n + k$. Alors les alignements (\bar{x}, \bar{y}) de (x, y) seront exactement les alignements construits en ajoutant $n + k - m$ gaps à y à des indices i tels que $\bar{x}_i \neq -$.

Les alignements sont donc en bijection avec les parties à $n + k - m$ éléments de $[1, n]$: il y en a C_{n+k-m}^n , en notant qu'on a bien $0 \leq n + k - m \leq n$.

Soit A_k l'ensemble des alignements produits en ajoutant k gaps à x , et A l'ensemble des alignements de (x, y) . On a :

$$|A| = \sum_{i=0}^m |A_i| = \sum_{i=0}^m C_{n+i}^i * C_{n+i}^{n+i-m}$$

Pour $|x| = 15$ et $|y| = 10$, on calcule 298, 199, 265 alignements.

Question 5 :

Soit la complexité de énumérer un alignement de (x, y) est $O(1)$.

Le nombre de alignement de (x, y) est $\sum_{i=0}^m C_{n+i}^i * C_{n+i-m}^n$, avec $|x| = n$ et $|y| = m$.

$$|A| = \sum_{i=0}^m C_{n+i}^i * C_{n+i-m}^n = \sum_{i=0}^m \frac{(n+i)!}{n!i!} * \frac{(n+i)!}{m!(n+i-m)!}$$

Donc, la complexité est en $O(m * n!)$

Et pour déduire un alignement de coût minimal, on a besoin de rechercher le nombre minimal dans la liste de coût des alignements, qui a la même complexité d'énumération.

Question 6 :

Estimation de complexité spatiale :

Selon l'algorithme posé dans le sujet, on peut déduire que le nombre de programme à traiter au maximum (dans le pire cas) est $n+m$. En même temps, il y a 6 variables stockées dans chaque itération de boucle. On suppose que la complexité en mémoire pour les 6 variables est en $O(k)$, donc en totale, la complexité spatiale est $O(k(n + m))$.

Évaluer la performance :

Selon la résultat qu'on obtenu, la taille d'instance au maximum est 12.

```
(base) zhourunlin@zhoudemacbook-pro codes % python3 testTache1.py ]
distance de TATATGAGTC et TATTT : 10
Time taken for execute dist_naif : 0.03053903579711914
distance de TGGGTGCTAT et GGGGTTCTAT : 8
Time taken for execute dist_naif : 5.945109605789185
distance de AACTGTCTTT et AACTGTTTT : 2
Time taken for execute dist_naif : 2.3081109523773193
distance de CTGGAAGTGCG et CTGAACTGG : 9
Time taken for execute dist_naif : 10.387193202972412
distance de GCTTAACTAACGA et GCATAACTCAGA : 9
Time taken for execute dist_naif : 432.54401898384094
```

Ensuit, on exécute la commande `./python3 Tache1.py & (hangup)` pour voir la consommation de mémoire dans **Moniteur d'activité**

Nom du processus	Mémoire	Threads	Ports	PID	Utilisateur
python3.9	47,9 Mo	2	27	1107	zhourunlin

Programmation dynamique

Question 7 :

Comme $\pi(\bar{u}) = x$ et $\pi(\bar{v}) = y$, on a $\bar{u}_l \in \{-, x_i\}$ et $\bar{v}_l \in \{-, v_i\}$. En particulier,

- si $\bar{u}_l = -$, alors $\bar{v}_l = y_j$;
- si $\bar{v}_l = -$, alors $\bar{u}_l = x_i$ et
- si u_l et v_l sont différents de $-$ alors ils valent respectivement x_i et y_j .

Question 8 :

On note $u = (\bar{u}_i)_{i=1}^{l-1}$ et $v = (\bar{v}_i)_{i=1}^{l-1}$

- Si $\bar{u}_l = -$, alors $C(\bar{u}, \bar{v}) = c_{ins} + C(u, v)$.
- Si $\bar{v}_l = -$, alors $C(\bar{u}, \bar{v}) = c_{dél} + C(u, v)$.
- Sinon, $C(\bar{u}, \bar{v}) = c_{sub}(\bar{u}_l, \bar{v}_l) + C(u, v)$.

Question 9 :

Comme les distances sont positives, par croissance de la somme il suffit d'optimiser la distance de manière gloutonne en appliquant une transformation de coût minimal au dernier caractère. Plus précisément :

- Si $i = 0$ ou $j = 0$ alors $D(i, j) = 0$
- Sinon
 - Si $x_i = y_j$ alors $D(i, j) = D(i-1, j-1)$ (comme $c_{sub}(x_i, y_j) = 0$, le meilleur choix est fait en laissant les deux caractères).
 - Sinon, $D(i, j) = \min\{c_{dél} + D(i-1, j), c_{ins} + D(i, j-1), c_{sub}(x_i, y_j) + D(i-1, j-1)\}$. Comme on ne peut pas mettre de gap au même endroit, on a le choix entre consommer x_i (dans quel cas on place un gap à y , induisant le coût $c_{dél}$)

et consommer y_j (dans quel cas on place un gap à x , induisant le coût c_{ins}) ou consommer x_i et y_j (cas traité à l'étape précédente).

Plus généralement, on a la relation de récurrence $C(i, j) = \min\{c_{del} + D(i-1, j), c_{ins} + D(i, j-1), c_{sub}(x_i, y_j) + D(i-1, j-1)\}$.

Question 10 :

On a $D(0, 0) = d(\varepsilon, \varepsilon)$. Par la borne supérieure sur la longueur des alignements trouvée plus haut, le seul alignement de $(\varepsilon, \varepsilon)$ est vide, donc de coût nul : $D(0, 0) = 0$.

Question 11 :

Soit $u \in \Sigma^*$ de longueur j . Par un argument sur les longueurs, le seul alignement possible de (ε, u) (respectivement (u, ε)) est $(-, u)$ (respectivement $(u, -)$). On a ainsi $C(0, j) = j c_{ins}$ et $C(i, 0) = i c_{del}$.

Question 12 :

```
def DIST_1(x, y):
    n <- |X|
    m <- |Y|
    return DIST_1_REC(x, y, i, j)

def DIST_1_REC(x, y, n, m):
    if i=0 and j=0, then:
        return 0

    if i=0, then:
        return j * C_ins

    if j=0, then:
        return i * C_del

    a <- DIST_1_REC(x, y, i-1, j) + C_ins
    b <- DIST_1_REC(x, y, i, j-1) + C_del
    c <- DIST_1_REC(x, y, i-1, j-1) + COUT(x, y, i, j)
    return min{a, b, c}

def COUT(x, y, i, j):
    #le cout de l'operation substitution
    if x[i] = y[j], then:
        return 0

    if ( (x[i]=A and y[j]=T) or
        (x[i]=T and y[j]=A) or
        (x[i]=G and y[j]=C) or
        (x[i]=C and y[j]=G) ), then:
        return 3

    return 4
```

Question 13 :

Le programme se termine en cas de $i = 0$ ou $j = 0$, donc le nombre d'itération est $i + j$ au maximum. En conséquence, la complexité de spatiale est en $O(i + j)$.

Question 14 :

On peut présenter la fonction **DIST_10** en forme de $A(n)$, avec $A(n)$ est la nombre de nœuds dans l'arbre récursive.

Donc, on a : $A(i+j) = 2A(i+j-1) + A(i+j-2) + 1$

On introduit $A'(n)$, et on peut déduire que : $A'(i+j) = 2A'(i+j-1) + A'(i+j-2)$

Le polynôme caractéristique est : $r^2 - 2r - 1$, et la racine positive est : $r = 1 + \sqrt{2}$

Car chaque l'appelle a une complexité de $O(1)$. donc la complexité est $O((1 + \sqrt{2})^{i+j})$ en total.

Question 15 :

- Si $j > 0$ et $D(i, j) = D(i, j-1) + C_{ins}$, nous pouvons dire que $x_i = _$, cela signifie que nous effectuons une opération d'insertion ici. Alors $\forall (\bar{s}, \bar{t}) \in Al^*(i, j-1), (\bar{s} \cdot _, \bar{t} \cdot y_j) \in Al^*(i, j)$
- Si $i > 0$ et $D(i, j) = D(i-1, j) + C_{del}$, nous pouvons dire que $y_j = _$, cela signifie que nous effectuons une opération de suppression ici. Alors $\forall (\bar{s}, \bar{t}) \in Al^*(i-1, j), (\bar{s} \cdot x_i, \bar{t} \cdot _) \in Al^*(i, j)$
- Si $D(i, j) = D(i-1, j-1) + C_{sub}(x_i, y_j)$, nous pouvons dire que $x_i = _$ et $y_j = _$, cela signifie que nous effectuons une opération de substitution ici. Alors $\forall (\bar{s}, \bar{t}) \in Al^*(i-1, j-1), (\bar{s} \cdot x_i, \bar{t} \cdot y_j) \in Al^*(i, j)$

Question 16 :

```
def SOL_1(T, x, y):
    n <- |x|
    m <- |y|
    return SOL_1_REC(T, x, y, n, m)

def SOL_1_REC(T, x, y, i, j):
    if i=0 and j=0, then:
        return (" ", " ")

    if i=0, then:
        return ("_"*j, y[1, ..., j])

    if j=0, then:
        return (x[1, ..., i], "_"*i)

    a = T[i-1][j-1] + C_sub(i, j)
    b = T[i-1][j] + C_ins
    c = T[i][j-1] + C_del

    if a=T[i][j], then:
        al_x, al_y = SOL_1_REC(T, x, y, i-1, j-1)
        return (al_x + x[i], al_y + y[j])

    if b=T[i][j], then:
        al_x, al_y = SOL_1_REC(T, x, y, i-1, j)
        return (al_x + x[i], al_y + "_")

    al_x, al_y = SOL_1_REC(T, x, y, i, j-1)
    return (al_x + "_", al_y + y[j])
```

Question 17 :

Nous l'avons déjà déduit, la complexité de **DIST_10** est $O((1 + \sqrt{2})^{i+j})$. Comme **SOL_10** est similaire en structure au **DIST_10**, donc la complexité est $O((1 + \sqrt{2})^{i+j})$.

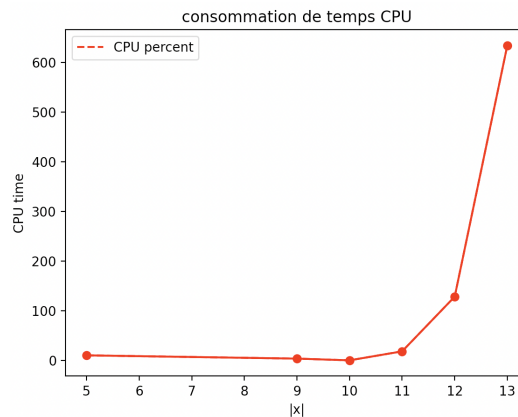
Pour la création de tableau, on a besoin de rappeler **DIST_10** $n * m$ fois, et après on crée le tableau, on exécute **SOL_10**. Donc, la complexité est $O(nm(1 + \sqrt{2})^{i+j} + (1 + \sqrt{2})^{i+j}) = O(nm(1 + \sqrt{2})^{i+j})$

Question 18 :

En combinant les algorithmes **DIST_10** et **SOL_10**, la complexité spatiale est la maximum de la complexité spatiale des deux algorithmes. En total, la complexité est $O(i + j)$.

Test de performance du code :

La bibliothèque *psutil* a été utilisée pour obtenir le pourcentage d'utilisation du CPU et de la mémoire pendant l'exécution du code. Après des pré-expériences, nous avons conclu que lorsque $|x|$ est supérieur à 13, l'exécution de **prog_dyn()** prendra plus de 10 minutes. Enfin, nous avons obtenu les traces suivantes sur les données qu'on obtenue :



Après tracer le graphe, on exécute les commandes sur le terminal avec $|x| = 10^3$ et 10^4 . Et on a :

Nom du processus	Mémoire	Threads	Ports	PID	Utilisateur
python3.9	50,5 Mo	2	27	3718	zhourunlin
python3.9	71,9 Mo	2	27	3719	zhourunlin

pid 3718: $|x| = 1000$; pid 3719: $|x| = 10000$;

Amélioration de la complexité spatiale du calcul de la distance

Question 19 :

Quand on a :

$T[i][j] = \min\{T[i-1][j] + C_{ins}, T[i][j-1] + C_{del}, T[i-1][j-1] + C_{sub}[i-1][j-1]\}$ donc quand on souhaite calculer $T[i][j]$, il suffit de prendre les valeurs dans des cases de coordonnées $(i, j-1)$, $(i-1, j-1)$, $(i-1, j)$ qui sont se situées dans des lignes i et $i-1$

Question 20 :

```
def DIST_2(x, y):
    n <- |x|, m <- |y|
    T <- tab[2][m+1] #T est un tableau vide
    for j de 0 à m+1, faire:
        T[0][j] <- j*c_ins

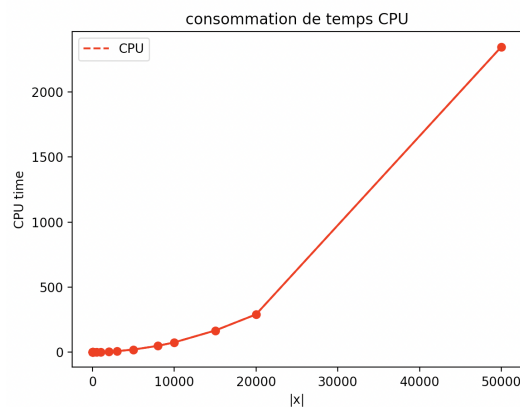
    for i de 1 à n+1, faire:
        T[1][0] <- i*c_del
        for j de 1 à m+1, faire:
            T[0][j] = min{ T[1][j]+C_ins,
                          T[0][j-1]+C_del,
                          T[0][j-1]+COUT(x, y, i-1, j-1) }

        T[0] <- T[1]

    return T[0][m]
```

Test de performance du code :

Après des pré-expériences, nous avons conclu que lorsque $|x|$ est supérieur à 5000 (avec s), l'exécution de **DIST_20** prendra plus de 10 minutes. Enfin, nous avons obtenu les traces suivantes sur les données qu'on obtienne :



Après tracer le graphe, on exécute les commandes sur le terminal avec $|x| = 10^3$, $5 * 10^3$, 10^4 , et $5 * 10^4$. Et on a :

Nom du processus	Mémoire	Threads	Ports	PID	Utilisateur
python3.9	61,3 Mo	2	27	10726	zhourunlin
python3.9	48,9 Mo	2	27	10731	zhourunlin
python3.9	48,9 Mo	2	27	10732	zhourunlin
python3.9	48,0 Mo	2	27	10737	zhourunlin

pid 10726: $|x| = 1000$; pid 10731: $|x| = 5000$; pid 10732: $|x| = 10000$; pid 10737: $|x| = 50000$;

Amélioration de la complexité spatiale du calcul d'un alignement optimal par la méthode "diviser pour régner"

Question 21 :

```
def mot_gaps(k):
    s <- ""
```

```
for i de 1 à k, faire:
    s <- s+"_"
return k
```

Question 22 :

```
def aligne_lettre_mot(x, y):
    coutMIN <- 4
    pos <- 1
    for i de 1 à |y|, faire:
        if C_sub(x[i], y[i]) < coutMIN, then:
            coutMIN <- C_sub(x[i], y[i])
            pos <- i
    xl <- "_"*(pos-1) + x[1] + "_"*(|y|-pos)
    return (xl, y)
```

Une extension : l'alignement local de séquences (Ex 4) :