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

> (i) $\pi(\overline{xu}) = xu$
> (ii) $\pi(\overline{yv}) = yv$
> (iii) $|\overline{xu}| = |\overline{yv}|$
> (iv) $\forall i \in [1, ..., |\overline{xu}|], \overline{xu}_i \neq - \ \ ou \ \ \overline{yv}_i \neq -$

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 $\forall i \in [1, ..., |\overline{x}|], \overline{x}_i \neq - \ \ ou \ \ \overline{y}_i \neq -$ et $\forall i \in [1, ..., |\overline{u}|], \overline{u}_i \neq - \ \ ou \ \ \overline{v}_i \neq -$, on ajoute les deux conditions emsemble. Et on a $\forall i \in [1, ..., |\overline{xu}|], \overline{xu}_i \neq - \ \ ou \ \ \overline{yv}_i \neq -$

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

Question 2 :

Si $(\overline{x}, \overline{y})$ est un alignment de (x, y), on a $\forall i \in [1, ..., |\overline{x}|], \overline{x}_i \neq - \ \ ou \ \ \overline{y}_1 \neq -$

donc la longueur maximale est $|x| + |y|$

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

Programmation naive :

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 $\overline{x}$ de longueur $n + k$. Alors les alignements $(\overline{x}, \overline{y})$ de (x, y) seront exactement les alignements construits en ajoutant $n + k - m$ gaps à y à des indices i tels que $\overline{x}_i \neq -$.
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 compelxité de énumérer un alignement de (x, y) est $O(1)$.

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

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

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

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

Question 6 :

Estimation de complexité spatiale :

Selon l'algorithme poser dans le sujet, on peut déduire que le nombre de programme à traiter au maxiume (dans le pire cas) est n+m. En même temp, il y a 6 variables stockées dans chaque inération 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. (avec GCTAACTAACG et GCTAAACTACT utilise 80s)

```
[(base) zhourunlin@zhoudemacbook-pro codes % python3 Tache1.py
Time taken for execution : 0.035063982009887695
distance de TATATGAGTC et TATTT : 10
Time taken for execution : 5.796966791152954
distance de TGGGTGCTAT et GGGGTTCTAT : 8
Time taken for execution : 2.2551279067993164
distance de AACTGTCTTT et AACTGTTTT 2
Time taken for execution : 10.18900203704834
distance de CTGGAAAGTGCG et CTGAACTGG : 9
Time taken for execution : 79.3547248840332
distance de GCTTAACTAACG et GCTAAACTACT : 9
```

Ensuit, on exécute la commande *./python3 Tache1.py &* (hangup), et la commande *top* pour voir la consommation mémoire (99.5% in the first line)

```
PID    COMMAND      %CPU  TIME     #TH   #WQ  #PORT MEM    PURG   CMPRS PGRP  PPID  STATE    BOOSTS          %CPU_ME %CPU_OTHRS UID
63042  python3.9    99.5  00:41.78 2/1   0    27    6785K  0B     0B    63042 63019 running  *0[1]           0.00000 0.00000    501
370    WindowServer 41.6  05:07:07 23    4    2775+ 1253M+ 12M+   305M  370   1     sleeping *0[1]           6.66574 1.03168    88
673    avconference 27.8  66:50.83 26    6    399   142M+  0B     14M   673   1     stuck    *0[1]           3.57124 15.55610   501
```

Programmation dynamique

Question 7 :

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

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

Question 8 :

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

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

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 $C(i,j) = min\{c_{dél} + D(i-1,j), c_{ins} + D(i,j-1), c_{sub}(x_i, y_j) + D(i-1,j-1)\}$.

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 $C(0,j) = jc_{ins}$ et $C(i,0) = ic_{dél}$.

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_dél

    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 CHOUT(x, y, i, j):
    if x[i] = y[j], then:
        return 0

    if ( (x[i]=A and y[i]=T) or
         (x[i]=T and y[i]=A) or
         (x[i]=G and y[i]=C) or
         (x[i]=C and y[i]=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_1()** 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 déduit que : $A'(i + j) = 2A'(i + j - 1) + A'(i + j - 2)$

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

Car chaque l'applle 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 $\forall(\overline{s}, \overline{t}) \in Al^*(i, j-1), (\overline{s} \cdot -, \overline{t} \cdot y_j) \in Al^*(i, j)$
- 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 $\forall(\overline{s}, \overline{t}) \in Al^*(i - 1, j), (\overline{s} \cdot x_i, \overline{t} \cdot -) \in Al^*(i, j)$
- 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 $\forall(\overline{s}, \overline{t}) \in Al^*(i - 1, j - 1), (\overline{s} \cdot x_i, \overline{t} \cdot y_j) \in Al^*(i, j)$

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-l, 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-l, j)
        return (al_x + x[i], al_y + "_")

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

Question 17 :

Nous l'avons déjà déduit, la complexité de **DIST_1()** est $O((1+\sqrt{2})^{i+j})$. Comme **SOL_1()** est similaire en structure au **DIST_1()**, donc la complexité est $O((1+\sqrt{2})^{i+j})$.

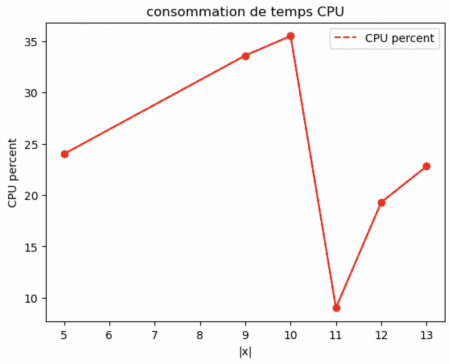
Pour la création de tableau, on a besoin de rappelle **DIST_1()** $n*m$ fois, et après on crée le tableau, on exécute **SOL_1().** Donc, la complexité est $O(nm(1+\sqrt{2})^{i+j}+(1+\sqrt{2})^{i+j}) = O(nm(1+\sqrt{2})^{i+j})$

Question 18 :

En combinant les algorithmes **DIST_1()** et **SOL_1()**, 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 execute la commande *top* sur le terminal avec $|x| = 10^3$, 10^4 et 10^5. Et on a :

```
Processes: 595 total, 4 running, 591 sleeping, 2792 threads            23:25:22
Load Avg: 2.35, 2.82, 2.51  CPU usage: 28.48% user, 3.84% sys, 67.66% idle
SharedLibs: 616M resident, 107M data, 35M linkedit.
MemRegions: 219734 total, 4704M resident, 378M private, 2511M shared.
PhysMem: 15G used (1723M wired), 215M unused.
VM: 235T vsize, 3823M framework vsize, 15921(0) swapins, 31697(0) swapouts.
Networks: packets: 26926119/29G in, 14693318/6669M out.
Disks: 5350263/84G read, 6160422/112G written.
```

```
Processes: 594 total, 4 running, 590 sleeping, 2779 threads            23:24:36
Load Avg: 2.32, 2.85, 2.51  CPU usage: 26.71% user, 3.12% sys, 70.15% idle
SharedLibs: 616M resident, 107M data, 35M linkedit.
MemRegions: 219350 total, 4692M resident, 379M private, 2567M shared.
PhysMem: 15G used (1750M wired), 253M unused.
VM: 235T vsize, 3823M framework vsize, 15921(0) swapins, 31697(0) swapouts.
Networks: packets: 26925341/29G in, 14692531/6669M out.
Disks: 5349183/84G read, 6158319/112G written.
```

```
Processes: 592 total, 4 running, 588 sleeping, 2700 threads            23:23:56
Load Avg: 3.00, 3.03, 2.55  CPU usage: 26.36% user, 2.66% sys, 70.97% idle
SharedLibs: 616M resident, 107M data, 35M linkedit.
MemRegions: 218932 total, 4669M resident, 375M private, 2515M shared.
PhysMem: 15G used (1708M wired), 323M unused.
VM: 234T vsize, 3823M framework vsize, 15921(0) swapins, 31697(0) swapouts.
Networks: packets: 26924871/29G in, 14692069/6668M out.
Disks: 5348869/84G read, 6157520/112G written.
```

Amélioration de la complexité spatiale du calcul de la distance

Question 19 :

Question 20 :

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

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

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

Question 21 :

```
det 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) :