

## Projet de LU3IN003

---

Danaël CARBONNEAU

### Alignement de séquences

Ce projet porte sur 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. Cela permet, lorsqu'un nouveau génome est séquencé, de comparer ses gènes à ceux de génomes précédemment séquencés, et de repérer ainsi des homologues, c'est-à-dire les ressemblances dues au fait que les deux espèces ont un ancêtre commun qui leur a transmis ce gène, même si ce gène a pu subir des mutations (évolutions) au cours du temps.

D'un point de vue informatique, les séquences de nucléotides sont vues comme des mots sur l'alphabet A,T,G,C et l'on est ramené à deux problèmes d'algorithmique du texte : le calcul de la distance d'édition entre deux mots quelconques et la production d'un alignement réalisant cette distance. Pour chacun de ces problèmes, on s'intéresse d'abord à un algorithme naïf, puis à un algorithme de programmation dynamique. Enfin, on utilise la méthode diviser pour régner pour améliorer la complexité spatiale de ces algorithmes. En ouverture, on s'intéresse à un problème légèrement différent : l'alignement local de séquences.

# Contents

<b>1</b>	<b>Présentation du code</b>	<b>4</b>
<b>2</b>	<b>Le problème d'alignement de séquences</b>	<b>6</b>
2.1	Alignement de deux mots . . . . .	6
<b>3</b>	<b>Algorithmes Pour l'alignement de séquences</b>	<b>8</b>
3.1	Méthode naïve par énumération . . . . .	8
3.2	Programmation dynamique . . . . .	12
3.3	Amélioration de la complexité spatiale du calcul de la distance . . . . .	18
3.4	Amélioration de la complexité spatiale par la méthode "diviser pour régner" . .	20
<b>4</b>	<b>Une extension : l'alignement local de séquences (Bonus)</b>	<b>24</b>
4.1	Approche théorique de cette extension . . . . .	24

## Présentation du code

Nous avons fait le choix d'implémenter l'ensemble des tâches demandées en C. Le code est rangé dans différents dossiers afin de faciliter son utilisation.

Des makefile servent à la compilation là où cela est nécessaire.

### Fonctions\_generales

Ce dossier répertorie les fonctions utilisées de manière générale par les autres fichiers.

#### structures

Pour mieux manipuler les chaînes et fichiers utilisés, nous avons utilisé différentes structures dont le principal objectif est l'accès rapide aux données essentielles et le retour de plusieurs valeurs par une même fonction.

**Couple\_chaine** : c'est une structure qui permet de retenir facilement nos couples de chaînes de caractères manipulés en les retenant les deux ainsi que leur longueur.

**Alignement** : cette structure surtout utile dans la partie concernant la programmation dynamique et contient au alignement de séquences de nucléotides, ainsi que sa distance d'édition, sa taille, et un curseur qui sert au remplissage dans **SOL\_1** puisqu'on remonte dans le tableau et commence donc au dernier caractère de  $(\bar{x}, \bar{y})$  (cela nous évite de devoir renverser les chaînes à la fin.

**Nom\_taille\_x** : cette structure nous permet de retenir à la fois le nom d'un fichier et la taille de x le contenant et sert dans le parcours du dossier contenant les instances, notamment pour pouvoir ensuite les trier par taille de x.

**Tableau\_fichiers** : cette structure contient un tableau de noms de fichiers (avec taille de x) ainsi que le nombre de fichiers lus, qui est utile dans le cas où le nombre de fichiers dans le dossier n'est pas assez grand pour le remplir intégralement.

#### Outils

Outils contient les constantes utilisées tout au long du projet ainsi que trois fonctions qui sont utiles peu importe la méthode utilisée pour résoudre le problème.

### Fonctions\_lecture\_ecriture

Ce code permet la lecture formatée de nos génomes.

## **Outils\_pour\_experimentation**

Ce dossier contient le code nécessaire à la manipulation du dossier utilisé pour les expérimentations sur de nombreuses instances.

## **Instances\_genome**

Il s'agit du dossier fourni contenant des exemples d'instances à traiter, nous en avons rajouté une pour la dernière partie du projet nommée "Instance\_x\_plus\_grand\_que\_y.adn".

## **Approche naïve**

Ce dossier contient le code correspondant à la partie 1 du projet.

## **Programmation dynamique**

Ce dossier contient les codes implémentant les trois tâches de cette partie du projet. Chaque Tâche est intégralement implémentée dans le dossier avec le nom correspondant. Des mains servent à tester rapidement le fonctionnement des fonctions.

## **Experimentations**

Ce dossier contient tous nos main permettant l'expérimentation de nos fonctions (test de consommation temporelle et mémoire).

Chaque main est fait avec un menu permettant de choisir les tâches à effectuer avant d'arrêter l'exécution.

### **Experimentation\_approche\_naive.c**

Main permettant de chercher la valeur minimale pour laquelle notre méthode s'effectue en moins d'une minute, ainsi que de la tester sur une valeur afin d'avoir la consommation mémoire.

### **Experimentation\_Programmation\_dynamique.c**

Main permettant de faire les expérimentations des tâche B et C

### **Experimentation\_Diviser\_pour\_mieux\_regner.c**

Main permettant de faire les expérimentations de la tâche D.

### **Extension\_alignement\_local**

Main nous permettant de faire des observations utiles pour la dernière partie.

# 2

## Le problème d'alignement de séquences

### Exercice 2.1 Alignement de deux mots

#### Q.2.1.1 Démonstrations de base sur les alignements

On cherche à montrer que si  $(\bar{x}, \bar{y})$  et  $(\bar{u}, \bar{v})$  sont des alignements de  $(x, y)$  et  $(u, v)$ , alors  $(\bar{x} \cdot \bar{u}, \bar{y} \cdot \bar{v})$  est un alignement de  $(x \cdot u, y \cdot v)$ .

Il va nous falloir pour cela montrer les quatre propriétés définissant un alignement :

1.  $\pi(\bar{x} \cdot \bar{u}) = x \cdot u$
2.  $\pi(\bar{y} \cdot \bar{v}) = y \cdot v$
3.  $|\bar{x} \cdot \bar{u}| = |\bar{y} \cdot \bar{v}|$
4.  $\forall i \in [1, \dots, |\bar{x} \cdot \bar{u}|], x_i \neq - \text{ ou } y_i \neq -$

**Montrons que**  $\pi(\bar{x} \cdot \bar{u}) = x \cdot u$

$\pi(\bar{x} \cdot \bar{u})$  est le sous mot obtenu en retirant tous les  $-$  de  $\bar{x} \cdot \bar{u}$ , il contient donc tous les caractères différents de  $-$  dans leur ordre d'apparition dans  $\bar{x}$ , soit  $x$  car  $\pi(\bar{x}) = x$ , puis tous les caractères différents de  $-$  dans leur ordre d'apparition dans  $\bar{u}$ , soit  $u$  car  $\pi(\bar{u}) = u$ , par définition de la fonction  $\pi$ , on retrouve alors la chaîne concaténée  $x \cdot u$ , et on a bien  $\pi(\bar{x} \cdot \bar{u}) = x \cdot u$

**Montrons que**  $\pi(\bar{y} \cdot \bar{v}) = y \cdot v$

De même,  $\pi(\bar{y} \cdot \bar{v})$  est le sous mot obtenu en retirant tous les  $-$  de  $\bar{y} \cdot \bar{v}$ , il contient donc tous les caractères différents de  $-$  dans leur ordre d'apparition dans  $\bar{y}$ , soit  $y$  car  $\pi(\bar{y}) = y$ , puis tous les caractères différents de  $-$  dans leur ordre d'apparition dans  $\bar{v}$ , soit  $v$  car  $\pi(\bar{v}) = v$ , par définition de la fonction  $\pi$ , on retrouve alors la chaîne concaténée  $y \cdot v$ , et on a bien  $\pi(\bar{y} \cdot \bar{v}) = y \cdot v$ .

**Montrons que**  $|\bar{x} \cdot \bar{u}| = |\bar{y} \cdot \bar{v}|$

$$|\bar{x} \cdot \bar{u}| = |\bar{x}| + |\bar{u}| \quad \text{car concaténé} \quad (2.1)$$

$$= |\bar{y}| + |\bar{v}| \quad \text{car alignements} \quad (2.2)$$

$$= |\bar{y} \cdot \bar{v}| \quad \text{Car concaténé} \quad (2.3)$$

**Montrons que**  $\forall i \in [1, \dots, |\bar{x} \cdot \bar{u}|], x_i \neq -$  **ou**  $y_i \neq -$

Prenons  $i \in [1, \dots, |\bar{x} \cdot \bar{u}|]$ , soit

- $x_i \in \bar{x}$ , et dans ce cas,  $y_i \in \bar{y}$  car  $|\bar{x}| = |\bar{y}|$ . Puisque  $(\bar{x}, \bar{y})$  est un alignement de  $(x, y)$ ,  $x_i \neq -$  ou  $y_i \neq -$ .
- $x_i \in \bar{u}$ , dans ce cas,  $y_i \in \bar{v}$  car le dernier caractère de  $\bar{y}$  se trouve au même indice que le dernier caractère de  $\bar{x}$ , puisque leurs normes sont identiques, un caractère de  $\bar{u}$  ne peut donc avoir le même indice qu'un caractère de  $\bar{y}$ , d'où  $y_i \in \bar{v}$ . Du fait que  $(\bar{u}, \bar{v})$  est un alignement de  $(u, v)$ ,  $x_i \neq -$  ou  $y_i \neq -$ .

Dans les deux cas, la propriété est vérifiée, du fait que les deux chaînes concaténées sont des alignements.

Les quatre propriétés de l'alignement sont donc vérifiées pour  $(\bar{x} \cdot \bar{u}, \bar{y} \cdot \bar{v})$  un alignement de  $(x \cdot u, y \cdot v)$ . Ainsi, on a bien que si  $(\bar{x}, \bar{y})$  et  $(\bar{u}, \bar{v})$  sont des alignements de  $(x, y)$  et  $(u, v)$ , alors  $(\bar{x} \cdot \bar{u}, \bar{y} \cdot \bar{v})$  est un alignement de  $(x \cdot u, y \cdot v)$ .

### Q.2.1.2 Longueur maximale d'un alignement

La longueur maximale d'un alignement de deux mots  $x$  et  $y$  de taille respectivement  $n$  et  $m$  est de  $n + m$ . Il est construit de tel sorte qu'à chaque caractère de  $x$ , le caractère aligné dans  $\bar{y}$  soit un  $-$ , et inversement. Par exemple, pour  $\Sigma = \{A, T, G, C\}$ ,  $x = ATGTA$  et  $y = GATTACA$ , on peut faire l'alignement de longueur  $n + m = 5 + 7 = 12$  suivant :

$$\begin{aligned}\bar{x} &= A - T - G - T - A - - - \\ \bar{y} &= -G - A - T - T - ACA\end{aligned}$$

Montrons que c'est la longueur maximale par l'absurde. Supposons qu'il existe un alignement de  $x, y \in \Sigma^*$  de longueur respectivement  $n$  et  $m$  tels que  $|\bar{x}| = |\bar{y}| = n + m + 1$ .

Il y aura donc, dans  $\bar{x}$ ,  $n + m + 1$  éléments, dont  $n$  qui seront les caractères de  $x$ , et  $m + 1$  qui seront des  $-$ . De même, dans  $\bar{y}$ , il y aura  $n + m + 1$  éléments dont  $m$  caractères de  $y$  et  $n + 1$  qui seront des  $-$ .

Or, si on construit cet alignement, il nous faudrait  $m + 1$  caractères dans  $\bar{y}$  pour que tous les  $-$  de  $\bar{x}$  soit alignés avec un  $y_i \neq -$ . Ne pouvant le faire, il y aura donc au moins un  $i \in [1, \dots, |\bar{x}|]$  tel que  $x_i = - = y_i$ , ce qui ne respecte pas la quatrième propriété des alignements, il y a donc une contradiction. De ce fait, il ne peut y avoir d'alignements d'une longueur supérieure à  $n + m$ .

# 3

## Algorithmes Pour l'alignement de séquences

### Exercice 3.1 Méthode naïve par énumération

#### Q.3.1.1 Dénombrement du nombre de mots en ajoutant k gaps à un mot

Sur un mot de taille  $n$ , pour placer les  $k$  gaps, on va chercher à combiner les positions de  $k$  éléments sur un ensemble de  $n + k$  positions. Ce qui correspond au coefficient binomial suivant:

$$\binom{n+k}{k} = C_{n+k}^k = \frac{(n+k)!}{k!(n+k-k)!}$$

#### Q.3.1.2 Nombre d'alignements possibles

Pour deux mots  $x$  et  $y$  de longueur  $n$  et  $m$  avec  $n \geq m$ , la longueur maximale de  $\bar{x}$  est  $n + m$  (comme vu en 1.1.2), avec  $k$  gaps qui ont donc une valeur strictement comprise entre 0 et  $m$ .

Pour  $k$  gaps ajoutés à  $x$  dans  $\bar{x}$ , il nous faut donc  $n + k - m$  gaps dans  $\bar{y}$  (Longueur du mot diminué nombre de caractères de  $y$ )

Pour obtenir le nombre possible d'alignements de  $(x, y)$ , nous allons procéder en trois étapes :

- Énumérer le nombre de combinaisons de gaps possibles dans  $\bar{y}$  pour un mot  $\bar{x}$  avec  $k_x$  gaps
- Multiplier ce nombre de combinaisons par le nombre de mots  $\bar{x}$  faisables avec  $k_x$  gaps
- Sommer ces combinaisons selon le nombre  $k_x \in [0; m]$  (plage de valeurs de  $k$  possible)

Prenons un exemple avec  $x = ACTG$ ,  $y = AG$  et  $k_x = 1$ . On a alors

$$\begin{aligned} k_y &= n + k_x - m \\ &= 4 + 1 - 2 \\ &= 3 \end{aligned}$$

Soit le mot  $\bar{x} = A - CTG$ , pour trouver les insertions de gap dans  $y$  possibles, on peut représenter le problème par un tableau de 5 colonnes, où la première ligne représente  $\bar{x}$  et les suivantes les différents mots  $\bar{y}$  associés :

A	-	C	T	G
-		-	A	G
-		A	-	G
-		A	G	-
A		-	-	G
A		-	G	-
A		G	-	-

Les cases colorées correspondent à la position où un gap ne peut se trouver dans y. Par extension de ce cas de base, on voit que, chaque position occupée par un gap dans  $\bar{x}$  ne pouvant être occupée par un gap dans  $\bar{y}$ , il s'agit d'une combinaison de  $k_y$  gaps parmi n positions possibles (celles correspondant à des caractères de x), c'est à dire :

$$\binom{n}{k_y} = \binom{n}{n+k_x-m}$$

On multiplie alors par le nombre de  $\bar{x}$  possibles pour un  $k_x$  donné, ce qui nous donne le nombre de façons d'insérer des gaps dans y de manière à respecter les contraintes d'alignement pour  $k_x$  gaps mis dans  $\bar{x}$  suivant :

$$\binom{n+k}{k_x} \times \binom{n}{n+k_x-m}$$

Le nombre d'alignements possibles de  $(x, y)$  se trouve alors en sommant le résultat précédent pour  $k_x = i$  allant de 0 à m:

$$\sum_{i=0}^m \binom{n+i}{i} \times \binom{n}{n+i-m}$$

Pour réaliser l'application numérique sur les valeurs  $|\bar{x}| = 15$  et  $|\bar{y}| = 10$ , nous allons utiliser un tableur (présent en fichier annexe) pour calculer les valeurs intermédiaires et leur somme. En résulte le tableau suivant :

$i$	$\binom{n+i}{i}$	$\binom{n}{n+i-m}$	$\binom{n+i}{i} \times \binom{n}{n+i-m}$
0	1	3003	3003
1	16	5005	80080
2	136	6435	875160
3	816	6435	5250960
4	3876	5005	19399380
5	15504	3003	46558512
6	54264	1365	74070360
7	170544	455	77597520
8	490314	105	51482970
9	1307504	15	19612560
10	3268760	1	3268760

On somme les éléments de la troisième colonne pour obtenir

$$\sum_{i=0}^{10} \binom{15+i}{i} \times \binom{15}{5+i} = 298199265$$



### Q.3.1.3 Complexité temporelle de notre méthode naïve

La complexité temporelle d'une méthode naïve qui consisterait à énumérer toutes les combinaisons possibles puis calculer la distance d'édition en vue de trouver un alignement de coût minimal serait alors :

- La complexité temporelle de création d'un alignement qu'on nommera  $c_a$
- Qu'on répéterait  $\sum_{i=0}^m \binom{n+i}{i} \times \binom{n}{n+i-m}$  fois pour obtenir tous les alignements possibles.
- Puis on parcourrait tous les alignements pour trouver leur distance d'édition tout en stockant, au fur et à mesure, le coût minimal<sup>1</sup>.

Le parcours seul des éléments énumérés se fait en  $O(m \times (n+m)!)$ , puisque le calcul du produit dans la somme faite sur  $m$  est un produit de coefficient binomiaux qui s'expriment à l'aide de factorielles<sup>2</sup>. De même pour la répétition de  $c_a$  pour obtenir tous les alignements. La complexité totale est alors en  $O(c_a \times (n+m)! + (n+m)!)$ . Ce qui est une complexité supérieure à du  $O(e^n)$ , c'est à dire pire qu'une complexité exponentielle. Il n'est donc pas réaliste de le penser utilisable.

### Q.3.1.4 Complexité spatiale de notre méthode naïve

Pour ce qui est de la complexité spatiale, nous nous baserons sur une structure de type tableau de tableaux de  $n+m$  cases où chaque case correspond à un caractère de  $(\bar{x}, \bar{y})$  et chaque ligne un alignement possible de  $(\bar{x}, \bar{y})$ .<sup>3</sup>

- Chaque case est composée de deux caractères (un octet chacun)
- Il y a au total  $\sum_{i=0}^m \binom{n+i}{i} \times \binom{n}{n+i-m}$  lignes dans notre matrice de dimension 2

La taille en mémoire de notre tableau d'alignements est donc de

$$c_{s_{\text{tableau}}} = 2 * \sum_{i=0}^m \binom{n+i}{i} \times \binom{n}{n+i-m}$$

Il nous faut en plus en mémoire un couple de valeurs max où un champ correspond au coût d'alignement minimal, et l'autre à un pointeur vers la première case de la ligne correspondant à l'alignement de coût minimal<sup>4</sup>. Cette structure de couple prend alors l'espace en mémoire d'un `int` et d'une adresse.

On a alors une complexité spatiale totale<sup>5</sup> de

$$c_s = 8 + 2 * \sum_{i=0}^m \binom{n+i}{i} \times \binom{n}{n+i-m} \text{octets}$$

Donc  $c_s \in O((n+m)!)$ , ce qui est une complexité, là encore, très importante.

---

<sup>1</sup>Le fait de chercher un coût minimal ne change rien à la complexité temporelle car il suffit de mettre à jour la valeur max après une comparaison en temps constant lors du calcul des coûts

<sup>2</sup> $(n+m)!$  majore le produit de nos coefficients binomiaux

<sup>3</sup>La liste chaînée pour représenter un alignement n'est pas spécialement avantageuse malgré le fait que sa taille soit inférieure ou égale à  $n+m$  car l'adresse de la case suivante augmente considérablement la taille de chaque cellule par rapport à un tableau.

<sup>4</sup>Considérant la taille du tableau, le coût d'accès à un élément à partir de son indice ne se fera pas forcément en  $\Theta(1)$  car il ne l'est pas sur des tableaux de taille trop importante, garder un pointeur directement sur la case n'est pas non plus un "gâchis" d'espace car l'indice pourra vite ne plus être stocké sur un `int`, ni même sur un `long` pour des valeurs de  $n$  et  $m$  peu élevées

<sup>5</sup>On fait la supposition d'une architecture en 32bits

## Tâche A

### Temps CPU

En testant notre fonction sur 10 instances<sup>6</sup> rangées par taille croissante de  $x$ , on obtient les temps suivants :

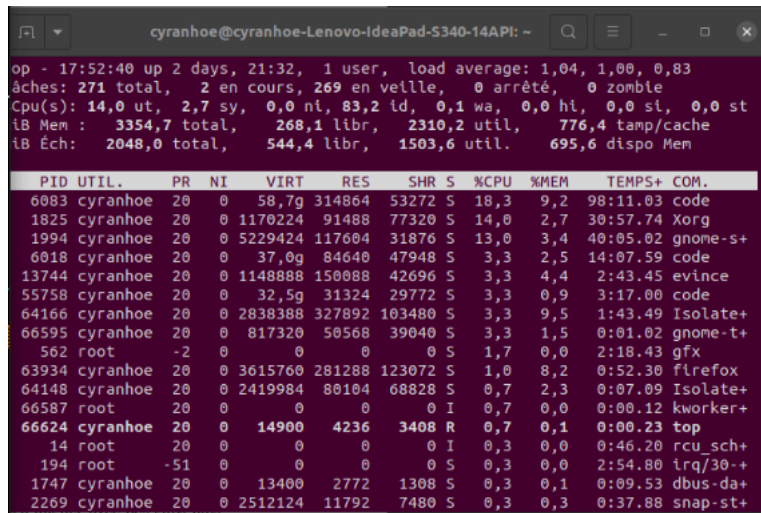
Instance	Temps (en secondes, deux chiffres significatifs)
Inst_0000010_8.adn	0.00
Inst_0000010_44.adn	0.00
Inst_0000012_32.adn	0.00
Inst_0000012_13.adn	0.00
Inst_0000012_56.adn	5.00
Inst_0000013_89.adn	21.00
Inst_0000013_45.adn	21.00
Inst_0000013_56.adn	22.00
Inst_0000014_23.adn	8.00
Inst_0000014_7.adn	61.00

On dépasse donc la minute pour l'instance `Inst_0000014_7.adn` , c'est à dire :

T G G G T G C T A T G T G C  
T T G G T G T A G T G C

### Mémoire

Concernant l'utilisation mémoire, en lançant la commande linux `top` après avoir lancé notre méthode sur l'instance `Inst_0000012_32.adn`, on obtient cet affichage :



```
cyranhoe@cyranhoe-Lenovo-IdeaPad-S340-14API: ~  
op - 17:52:40 up 2 days, 21:32, 1 user, load average: 1.04, 1.00, 0.83  
âches: 271 total, 2 en cours, 269 en veille, 0 arrêté, 0 zombie  
Cpu(s): 14,0 ut, 2,7 sy, 0,0 ni, 83,2 id, 0,1 wa, 0,0 hi, 0,0 si, 0,0 st  
ib Mem : 3354,7 total, 268,1 libr, 2310,2 util, 776,4 tamp/cache  
ib éch: 2048,0 total, 544,4 libr, 1503,6 util. 695,6 dispo Mem  
  
PID UTIL. PR NI VIRT RES SHR S %CPU %MEM TEMPS+ COM.  
6083 cyranhoe 20 0 58,7g 314864 53272 S 18,3 9,2 98:11.03 code  
1825 cyranhoe 20 0 1170224 91488 77320 S 14,0 2,7 30:57.74 Xorg  
1994 cyranhoe 20 0 5229424 117604 31876 S 13,0 3,4 40:05.02 gnome-s+  
6018 cyranhoe 20 0 37,0g 84640 47948 S 3,3 2,5 14:07.59 code  
13744 cyranhoe 20 0 1148888 150088 42696 S 3,3 4,4 2:43.45 evince  
55758 cyranhoe 20 0 32,5g 31324 29772 S 3,3 0,9 3:17.00 code  
64166 cyranhoe 20 0 2838388 327892 103480 S 3,3 9,5 1:43.49 Isolate+  
66595 cyranhoe 20 0 817320 50568 39040 S 3,3 1,5 0:01.02 gnome-t+  
562 root -2 0 0 0 0 S 1,7 0,0 2:18.43 gxf  
63934 cyranhoe 20 0 3615760 281288 123072 S 1,0 8,2 0:52.30 firefox  
64148 cyranhoe 20 0 2419984 80104 68828 S 0,7 2,3 0:07.09 Isolate+  
66587 root 20 0 0 0 0 I 0,7 0,0 0:00.12 kworker+  
66624 cyranhoe 20 0 14900 4236 3408 R 0,7 0,1 0:00.23 top  
14 root 20 0 0 0 0 I 0,3 0,0 0:46.20 rcu_sch+  
194 root -51 0 0 0 0 S 0,3 0,0 2:54.80 irq/30+  
1747 cyranhoe 20 0 13400 2772 1308 S 0,3 0,1 0:09.53 dbus-da+  
2269 cyranhoe 20 0 2512124 11792 7480 S 0,3 0,3 0:37.88 snap-st+
```

Figure 3.1: État de la mémoire en utilisant la méthode naive

La ligne 55758 nous indique alors que la consommation mémoire pour cette seule instance est de 0,9%, ce qui peut sembler peu, mais aux vues de la taille très petite de notre instance, il s'agit déjà d'une consommation conséquente.

<sup>6</sup>Notre test s'arrête quand le temps a dépassé les 60 secondes. Il ne dépend pas que de la taille de  $x$ , d'où le fait que ces instances ne soient pas rangées par l'ordre croissant de cette taille.

## Exercice 3.2 Programmation dynamique

### Calcul de la distance d'édition par programmation dynamique

#### Q.3.2.1 Alignements partiels

**Si  $\overline{u}_l = -$ , que vaut  $\overline{v}_l$  ?**

Puisqu'on ne peut avoir  $\overline{u}_l = -$  et  $\overline{v}_l = -$  (définition de l'alignement), et que  $\overline{v}_l$  contient tous les éléments de  $y_{[1...j]}$  sont dans  $\overline{v}_l$  dans leur ordre d'apparition dans  $y_{[1...j]}$ ,  $\overline{v}_l = y_j$  la dernière lettre de  $y_{[1...j]}$ .

**Si  $\overline{v}_l = -$ , que vaut  $\overline{u}_l$  ?**

De même, puisqu'on ne peut avoir  $\overline{u}_l = -$  et  $\overline{v}_l = -$  (définition de l'alignement), et que  $\overline{u}_l$  contient tous les éléments de  $x_{[1...i]}$  sont dans  $\overline{u}_l$  dans leur ordre d'apparition dans  $x_{[1...i]}$ ,  $\overline{u}_l = u_i$  la dernière lettre de  $u_{[1...i]}$ .

**Si  $\overline{u}_l \neq -$  et  $\overline{v}_l \neq -$ , que valent  $\overline{u}_l$  et  $\overline{v}_l$  ?**

$\overline{u}_l$  et  $\overline{v}_l$  étant les deux derniers éléments de  $(\overline{u}, \overline{v})$ , et puisqu'un alignement contient toutes les lettres de  $(x_{[1...i]}, y_{[1...j]})$ , alors, nécessairement,  $\overline{u}_l = x_i$  et  $\overline{v}_l = y_j$ .

#### Q.3.2.2 Trois cas envisagés

$$c(\overline{u}, \overline{v}) = c(\overline{u}_{[1...l-1]}, \overline{v}_{[1...l-1]}) + c(\overline{u}_l, \overline{v}_l)$$

Avec

$$c(\overline{u}_l, \overline{v}_l) = \begin{cases} c_{ins} & \text{si } \overline{u}_l = - \\ c_{del} & \text{si } \overline{v}_l = - \\ c_{sub}(x_i, y_j) & \text{sinon} \end{cases}$$

#### Q.3.2.3 Expression de D(i,j)

La distance d'édition jusqu'aux indices i et j correspondant à deux alignements partiels de longueur l correspond à l'alignement jusqu'à l'indice l - 1 auquel on ajoute ce qui a été fait pour l'indice l. On a alors la formule suivante :

$$\begin{aligned} Dist(i, j) &= d(x_{[1:i]}, y_{[1:j]}) \\ &= \min\{cas_{ins}, cas_{del}, cas_{sub}\} \end{aligned}$$

Où

$$\begin{cases} cas_{ins} &= Dist(i-1, j) + c_{ins} \\ cas_{del} &= Dist(i, j-1) + c_{del} \\ cas_{sub} &= Dist(i-1, j-1) + c_{sub}(x[i], y[j]) \end{cases}$$

#### Q.3.2.4 Cas nul pour DIST\_1

D(0,0) signifie la distance d'édition entre deux sous mots vides (on n'a ni avancé avec les deux indices dans  $x_{[1...i]}$  ou  $y_{[1...j]}$ ). La distance d'édition est donc nulle :

$$D(0,0) = 0$$

### Q.3.2.5 Cas de base de DIST\_1

**Que vaut  $D(0, j)$  pour  $j \in [1...m]$  ?**

Puisque  $i = 0$ , on a visité aucune lettre de x, d'autres termes, pour avoir un alignement partiel de longueur j, on n'a fait que des insertions dans y, représentées par un  $-$  dans  $\bar{x}_{[1...j]}$ . Ce qui signifie que le coût d'édition est unique (aucun choix possible) et ainsi  $D(0, j) = j \times c_{ins}$ .

**Que vaut  $D(i, 0)$  pour  $i \in [1...n]$  ?**

Puisque  $j = 0$ , on a visité aucune lettre de y, d'autres termes, pour avoir un alignement partiel de longueur j, on n'a fait que des suppressions dans y, représentées par un  $-$  dans  $\bar{y}_{[1...i]}$ . Ce qui signifie que le coût d'édition est unique (aucun choix possible) et ainsi  $D(i, 0) = i \times c_{del}$ .

### Q.3.2.6 Pseudo code de DIST\_1

---

**Algorithm 1** DIST\_1

---

**Require:** x,y deux mots de taille n et m

```
Distances[n+1][m+1]
i, j
for i in range [0;n] do
  for j in range [0;m] do
    if i = 0 then
      if j = 0 then
        Distances[i][j] = 0
      else
        Distances[i][j] = j × cins
      end if
    else
      if j = 0 then
        Distances[i][j] = i × cdel
      else
        cas_ins = Dist(i, j - 1) + cins
        cas_del = Dist(i - 1, j) + cdel
        cas_sub = Dist(i - 1, j - 1) + csub(x[i], y[j])
        Distances[i][j] = min{cas_ins, cas_del, cas_sub}
      end if
    end if
  end for
end for
return Distances[n][m]
```

---

### Q.3.2.7 Complexité spatiale de DIST\_1

Il faut stocker un tableau de taille  $n \times m^7$  en mémoire, contenant des entiers, d'où une complexité spatiale en  $\Theta(n \times m)$

---

<sup>7</sup>n la taille de x et m la taille de y

### Q.3.2.8 Complexité temporelle de DIST\_1

On itère sur deux boucles imbriquées, la première allant de 0 à n et la seconde allant de 0 à m (parcourt du tableau pour renseigner la valeur de chaque case). Les opérations à l'intérieur de la seconde boucle sont des comparaisons et des affectations, qui se font en temps constant. En résulte une complexité temporelle en  $\Theta(n \times m)$ .

### Calcul d'un alignement optimal par programmation dynamique

Pour  $i \in [0..n]$  et  $j \in [0..m]$ , on ajoute aux notations précédentes une notation pour l'ensemble des alignements optimaux :

$Al * (i, j) = (\bar{u}, \bar{v}) | (\bar{u}, \bar{v})$  est un alignement de  $(x_{[1..i]}, y_{[1..j]})$  tel que  $C(\bar{u}, \bar{v}) = d(x_{[1..i]}, y_{[1..j]})$

### Q.3.2.9 Formules récursives sur la formation d'un alignement de coût minimum à partir d'un tableau de distances d'édition

On cherche à montrer que

si  $j > 0$  et  $D(i, j) = D(i, j - 1) + c_{ins}$ , alors  $\forall (\bar{s}, \bar{t}) \in Al * (i, j - 1), (\bar{s} \cdot -, \bar{t} \cdot y_j) \in Al * (i, j)$

Pour passer de  $D(i, j - 1)$  à  $D(i, j)$ , on a fait de nouveaux alignements contenant cette fois un élément de y supplémentaire (noté  $y_j$ ). L'alignement qui nous intéresse est celui de coût minimal, c'est à dire que son coût d'édition correspond à  $D(i, j)$ , or, puisque  $D(i, j) = D(i, j - 1) + c_{ins}$ , le nouvel alignement contenant  $y_j$  a été fait en faisant une insertion à un alignement  $Al * (i, j - 1)$ . En le notant  $(\bar{s}, \bar{t})$ , puisqu'on y a inséré  $y_j$ , on se retrouve alors avec un alignement  $(\bar{s} \cdot -, \bar{t} \cdot y_j)$  qui correspond au coût minimal  $D(i, j)$ , c'est à dire que  $(\bar{s} \cdot -, \bar{t} \cdot y_j) \in Al * (i, j)$ .

De même pour montrer que

si  $i > 0$  et  $D(i, j) = D(i - 1, j) + c_{del}$ , alors  $\forall (\bar{s}, \bar{t}) \in Al * (i, j - 1), (\bar{s} \cdot x_i, \bar{t} \cdot -) \in Al * (i, j)$

Pour passer de  $D(i - 1, j)$  à  $D(i, j)$ , on a fait de nouveaux alignements contenant cette fois un élément de x supplémentaire (noté  $x_i$ ). L'alignement qui nous intéresse est celui de coût minimal, c'est à dire que son coût d'édition correspond à  $D(i, j)$ , or, puisque  $D(i, j) = D(i - 1, j) + c_{del}$ , le nouvel alignement contenant  $y_j$  a été fait en faisant une insertion à un alignement  $Al * (i - 1, j)$ . En le notant  $(\bar{s}, \bar{t})$ , puisqu'on y a inséré  $x_i$ , on se retrouve alors avec un alignement  $(\bar{s} \cdot x_i, \bar{t} \cdot -)$  qui correspond au coût minimal  $D(i, j)$ , c'est à dire que  $(\bar{s} \cdot x_i, \bar{t} \cdot -) \in Al * (i, j)$ .

Enfin, pour montrer que

si  $i > 0, j > 0$  et  $D(i, j) = D(i - 1, j - 1) + c_{sub}(x_i, y_j)$ , alors  $\forall (\bar{s}, \bar{t}) \in Al * (i - 1, j - 1), (\bar{s} \cdot x_i, \bar{t} \cdot y_j) \in Al * (i, j)$

Pour passer de  $D(i - 1, j - 1)$  à  $D(i, j)$ , on a fait de nouveaux alignements contenant cette fois un élément de x supplémentaire (noté  $x_i$ ) et un élément de y supplémentaire (noté  $y_j$ ). L'alignement qui nous intéresse est celui de coût minimal, c'est à dire que son coût d'édition correspond à  $D(i, j)$ , or, puisque  $D(i, j) = D(i - 1, j - 1) + c_{sub}(x_i, y_j)$ , le nouvel alignement contenant  $x_i$  et  $y_j$  a été fait en faisant une substitution à un alignement  $Al * (i - 1, j - 1)$ . En le notant  $(\bar{s}, \bar{t})$ , puisqu'on y a inséré  $x_i$  et  $y_j$ , on se retrouve alors avec un alignement  $(\bar{s} \cdot x_i, \bar{t} \cdot y_j)$  qui correspond au coût minimal  $D(i, j)$ , c'est à dire que  $(\bar{s} \cdot x_i, \bar{t} \cdot y_j) \in Al * (i, j)$ .

### Q.3.2.10 Pseudo code de la fonction SOL\_1

L'algorithme parcourt le tableau D en y trouvant un chemin suivant la construction d'un alignement de coût minimal. On utilise pour cela les résultats obtenus à la question précédente.

---

#### Algorithm 2 SOL\_1

---

**Require:**  $x, y$  deux mots de taille  $n$  et  $m$ , D, un tableau de taille  $(n + 1) \times (m + 1)$  contenant les distances d'édition  $D(i, j)$  pour  $0 \leq i \leq n$  et  $0 \leq j \leq m$   
 $i = n, j = m$   
 $\bar{x}, \bar{y}$  {deux chaînes de caractère}  
**while**  $i > 0$  or  $j > 0$  **do**  
  **if**  $j > 0$  and  $D(i, j) = D(i, j - 1) + c_{ins}$  **then**  
     $\bar{x} = - \cdot \bar{x}$   
     $\bar{y} = y_j \cdot \bar{y}$   
     $j = j - 1$   
  **else if**  $i > 0$  and  $D(i, j) = D(i - 1, j) + c_{del}$  **then**  
     $\bar{x} = x_i \cdot \bar{x}$   
     $\bar{y} = - \cdot \bar{y}$   
     $i = i - 1$   
  **else if**  $i > 0$  et  $j > 0$  et  $D(i, j) = D(i - 1, j - 1) + c_{sub}(x_i, y_j)$  **then**  
     $\bar{x} = x_i \cdot \bar{x}$   
     $\bar{y} = y_j \cdot \bar{y}$   
     $i = i - 1$   
     $j = j - 1$   
  **else**  
    afficher ("Erreur dans D")  
     $\bar{x} = ' E' \cdot \bar{x}$   
     $\bar{y} = ' E' \cdot \bar{y}$   
    **return**  $(\bar{x}, \bar{y})$   
  **end if**  
**end while**  
**return**  $(\bar{x}, \bar{y})$

---

### Q.3.2.11 Complexité de SOL\_1

SOL\_1 parcourt un tableau de dimension  $n \times m$  en partant de la dernière case du tableau et en reculant alors soit vers la gauche, soit vers le haut, soit en diagonale. Ainsi, le plus grand nombre d'itérations possibles de notre boucle while est  $n + m$ . Il s'agit d'un parcours correspondant à  $n$  suppressions et  $m$  insertions (c'est à dire un alignement où  $\forall \bar{x}_i$  tel que si  $\bar{x}_i \neq -$  alors  $\bar{y}_i = -$  et inversement). En implémentant l'algorithme avec une structure qui permet les ajouts en tête dans une chaîne de caractères (on commence par le dernier élément de  $(\bar{x}, \bar{y})$ , en temps constant, on a alors que la complexité de SOL\_1 est donc en  $O(n + m)$ . La complexité de DIST\_1 étant en  $\Theta(n \times m)$ , et puisqu'il faut simplement exécuter un algorithme puis l'autre, on a alors une complexité en  $O(n \times m + n + m) = \Theta(n \times m)$ ,  $m$  étant toujours entier et positif,  $n + m$  sera toujours négligeable vis à vis de  $n \times m$ .

### Q.3.2.12 Complexité spatiale de SOL\_1

La complexité spatiale de l'implémentation de SOL\_1 va dépendre de la structure utilisée en mémoire pour représenter notre couple de chaînes de caractères. On pourrait soit

- utiliser un tableau de caractères de taille fixe  $n + m$ , qui correspond à la taille maximale de l'alignement, c'est à dire une complexité spatiale en  $\Theta(n + m)$ , il faudrait alors garder un curseur dans le tableau pour faire un ajout en  $\Theta(1)$
- utiliser une liste chaînée de caractères, dont la taille varie mais ne dépasse pas  $n + m$ , on aurait donc une complexité spatiale en  $O(n + m)$  et en  $\Omega(\max(n, m))$ . La complexité spatiale sera alors bien meilleure sur de grandes valeurs de  $n$  et  $m$  pour des alignements présentant beaucoup de substitutions.

Pour des raisons de simplicité, et parce que DIST\_1 a une complexité spatiale en  $\Theta(n \times m)$  (ce qui veut dire qu'une complexité spatiale en  $n + m$  devient négligeable à côté), nous utiliserons deux tableaux de caractères de taille  $n + m$  pour représenter en mémoire  $(\bar{x}, \bar{y})$ .

Notre solution requiert donc :

- Un tableau de distances d'édition de taille  $(n + 1) \times (m + 1)$
- Deux tableaux de caractères représentant  $x$  et  $y$  de taille respectivement  $n$  et  $m$
- Deux tableaux de caractères représentant  $\bar{x}$  et  $\bar{y}$  de taille  $n + m$

Notre complexité spatiale est donc en  $\Theta(n \times m + n + m + 2(n + m)) = \Theta(n \times m)$ .

## Tâche B

### Temps CPU

8

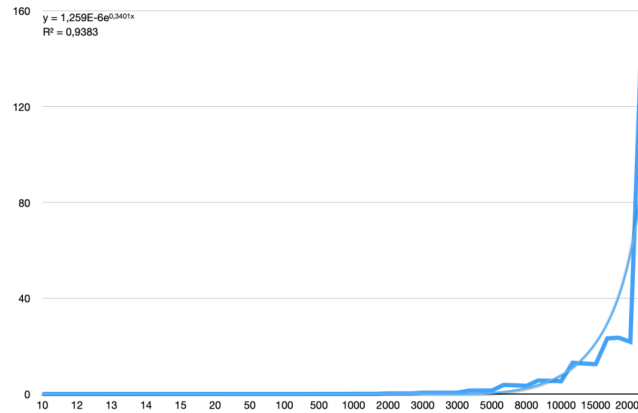


Figure 3.2: Consommation temporelle de `prog_dyn` fonction de la taille de x

On voit que la consommation temporelle réelle ne correspond pas à la formule calculée : elle semble rester nulle très longtemps, puis croître exponentiellement pour de très grandes valeurs : cela peut s'expliquer par le fait que jusqu'à ces très grandes valeurs, la consommation temporelle est faible et linéaire, mais à partir du moment où les instances deviennent trop grandes, les tableaux également, et un accès mémoire sur un tableau de taille très élevée ne se fait pas, en réalité, en temps constant, d'où l'écart entre la complexité théorique et ce qui s'observe de manière pratique.

### Mémoire

On a arbitrairement choisi l'instance `Inst_0015000_20.adn` pour faire nos essais de consommation mémoire avec la commande linux `top`.

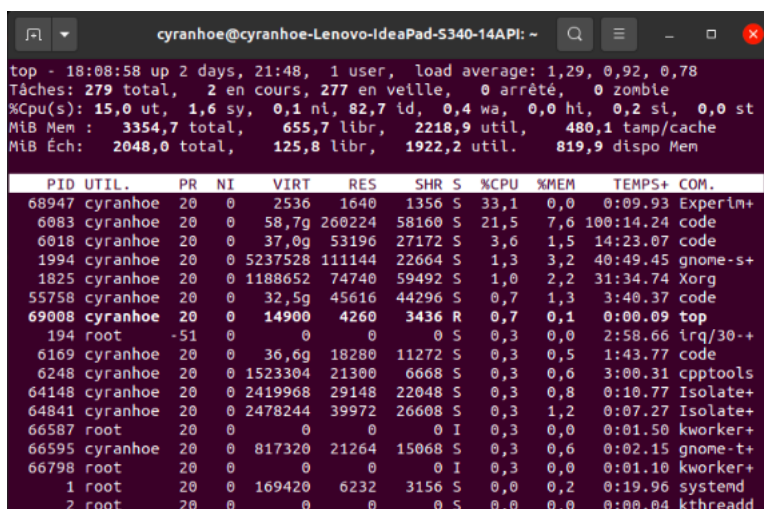


Figure 3.3: Consommation mémoire de `prog_dyn`

La ligne 6083 correspond à l'exécution de `prog_dyn`, qui nous indique 7,6% de consommation mémoire, ce qui est assez important, mais cohérent avec nos calculs.

<sup>8</sup>Il s'agit de la seule courbe que nous avons réussi à tracer, grâce à l'emprunt d'un autre ordinateur



## Exercice 3.3 Amélioration de la complexité spatiale du calcul de la distance

### Q.3.3.1 Ligne nécessaire au remplissage de la ligne i

À chaque itération dans `DIST_1`, on n'utilise que les indices  $i$  et  $i - 1$  ainsi que  $j$  et  $j - 1$ , qui sont donc tous contenus dans les deux lignes `D[i]` et `D[i-1]`. On peut donc écrire un algorithme qui n'utilise que ces deux lignes pour calculer la distance d'édition<sup>9</sup>

### Q.3.3.2 Pseudo code de la fonction `DIST_2`

---

#### Algorithm 3 `DIST_2`

---

**Require:**  $x, y$  deux mots de taille  $n$  et  $m$ , Distances un tableau de taille  $2 \times n$ <sup>10</sup>

```
i, j
for i in range [0;n] do
  for j in range [0;m] do
    if i = 0 then
      if j = 0 then
        Distances[0][j] = 0
      else
        Distances[0][j] =  $j \times c_{ins}$ 
      end if
    else
      if j = 0 then
        Distances[1][j] =  $i \times c_{del}$ 
      else
        cas_ins =  $Distances(1, j - 1) + c_{ins}$ 
        cas_del =  $Distances(0, j) + c_{del}$ 
        cas_sub =  $Distances(0, j - 1) + c_{sub}(x[i], y[j])$ 
        Distances[1][j] =  $\min\{cas\_ins, cas\_del, cas\_sub\}$ 
      end if
    end if
  end for
end for
for j in range [0;m] do
  Distances[0][j] = Distances[1][j]
end for
end for
return Distances[1][m]
```

---

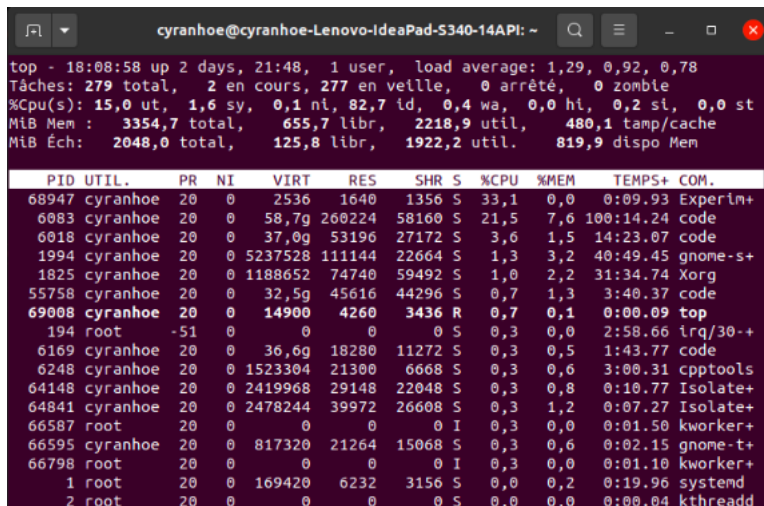
<sup>9</sup>Cependant, on a besoin de tout `dist` pour utiliser `SOL_1`, `Dist_2` ne nous donnera que la distance d'édition, et ne peut être utilisé pour retourner un alignement de coût minimal avec `SOL_1`.

## Tâche C

*On n'a pas pu tracer les courbes de consommation temporelle car les valeurs étaient soit nulles soient incalculables sans que l'ordinateur ne plante.*

### Mémoire

On obtient en lançant la commande linux top sur la même instance qu'à la tâche B que la consommation mémoire est bien plus faible : la ligne 6018 correspondante nous renseigne que nous n'utilisons désormais plus que 1,5% de la mémoire.



```
top - 18:08:58 up 2 days, 21:48, 1 user, load average: 1,29, 0,92, 0,78
Tâches: 279 total, 2 en cours, 277 en veille, 0 arrêté, 0 zombie
%Cpu(s): 15,0 ut, 1,6 sy, 0,1 ni, 82,7 id, 0,4 wa, 0,0 hi, 0,2 si, 0,0 st
MiB Mem : 3354,7 total, 655,7 libr, 2218,9 util, 480,1 temp/cache
MiB Éch: 2048,0 total, 125,8 libr, 1922,2 util, 819,9 dispo Mem
```

PID	UTIL.	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TEMPS+	COM.
68947	cyranhoe	20	0	2536	1640	1356	S	33,1	0,0	0:09.93	Experim+
6083	cyranhoe	20	0	58,7g	260224	58160	S	21,5	7,6	100:14.24	code
6018	cyranhoe	20	0	37,0g	53196	27172	S	3,6	1,5	14:23.07	code
1994	cyranhoe	20	0	5237528	111144	22664	S	1,3	3,2	40:49.45	gnome-s+
1825	cyranhoe	20	0	1188652	74740	59492	S	1,0	2,2	31:34.74	Xorg
55758	cyranhoe	20	0	32,5g	45616	44296	S	0,7	1,3	3:40.37	code
69008	cyranhoe	20	0	14900	4260	3436	R	0,7	0,1	0:00.09	top
194	root	-51	0	0	0	0	S	0,3	0,0	2:58.66	irq/30-+
6169	cyranhoe	20	0	36,6g	18280	11272	S	0,3	0,5	1:43.77	code
6248	cyranhoe	20	0	1523304	21300	6668	S	0,3	0,6	3:00.31	cpptools
64148	cyranhoe	20	0	2419968	29148	22048	S	0,3	0,8	0:10.77	Isolate+
64841	cyranhoe	20	0	2478244	39972	26608	S	0,3	1,2	0:07.27	Isolate+
66587	root	20	0	0	0	0	I	0,3	0,0	0:01.50	kworker+
66595	cyranhoe	20	0	817320	21264	15068	S	0,3	0,6	0:02.15	gnome-t+
66798	root	20	0	0	0	0	I	0,3	0,0	0:01.10	kworker+
1	root	20	0	169420	6232	3156	S	0,0	0,2	0:19.96	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.04	kthread

Figure 3.4: Consommation mémoire de prog\_dyn

## Exercice 3.4 Amélioration de la complexité spatiale par la méthode "diviser pour régner"

### Q.3.4.1 Pseudo code de la fonction `mot_gaps`

---

**Algorithm 4** `mot_gaps`

---

**Require:** `k` un entier  
    `res[k]` chaîne de caractères qu'on va remplir de `k` gaps  
    `i` notre indice de parcours  
    **for** `i` in range `[0;k[` **do**  
        `res[i] = '-'`  
    **end for**  
    **return** `res`

---

### Q.3.4.2 Pseudo code de la fonction `align_lettre_mot`

---

**Algorithm 5** `align_lettre_mot`

---

**Require:** `x` une lettre, `y` un mot de longueur `m`  
     $(\bar{x}, \bar{y})$  un alignement de taille  $m + 1$   
    `i` l'indice de parcours de l'alignement, initialisé à 0  
    `placex` l'indice où insérer `x` dans l'alignement, initialisé à -1 (pour dire qu'on n'a pas placé `x`).  
    `yj` une lettre de `y`.  
    **for** `yj` dans `y` **do**  
        **if**  $c_{sub}(x, y_j)$  est le coût d'opération le plus faible **then**  
            `placex = i`  
        **end if**  
        `i = i + 1`  
    **end for**  
    **if** `placex = -1` **then**  
         $\bar{x}[0] = x$   
         $\bar{y}[0] = '-'$   
         $\bar{x}[1; m] = \text{mot\_gaps}(m)$   
         $\bar{x}[1; m] = y$   
    **else**  
         $\bar{x}[\text{place}_x] = x$   
         $\bar{x}[0; \text{place}_x - 1] = \text{mot\_gaps}(\text{place}_x - 1)$   
         $\bar{x}[\text{place}_x + 1; m - 1] = \text{mot\_gaps}(m - \text{place}_x - 2)$   
         $\bar{y} = y$   
    **end if**  
    **return**  $(\bar{x}, \bar{y})$

---

### Q.3.4.3 Montrer que $(\bar{s} \cdot \bar{u}, \bar{t} \cdot \bar{v})$ n'est pas un alignement optimal de $(x, y)$

Un alignement de  $(x^1, y^1)$  optimal est  $(BAL, RO-)$  de distance d'édition 13 (on aligne les voyelles et les consonnes ensemble et on complète avec des -). Un alignement optimal de  $(x^2, y^2)$  est  $(LON-, - - ND)$  de distance d'édition 9.

D'où

$\overline{x^1} \cdot \overline{x^2}$	B	A	L	L	O	N	—
$\overline{Y^1} \cdot \overline{Y^2}$	R	O	—	—	—	N	D

Son coût est de 22, pourtant un alignement optimal de  $(x, y)$  est

$\overline{x}$	B	A	L	L	O	N	—
$\overline{y}$	R	—	—	—	O	N	D

La distance d'édition est alors 17. On voit alors un contre-exemple qui nous prouve que  $(\overline{s} \cdot \overline{u}, \overline{t} \cdot \overline{v})$  n'est pas un alignement optimal de  $(x, y)$ .

#### Q.3.4.4 Pseudo code de la fonction SOL\_2

---

##### Algorithm 6 SOL\_2

---

**Require:**  $x_1, x_2$ , deux séquences à aligner, de taille respectivement  $n$  et  $m$

```

if  $n = 0$  then
    return (mots_gaps(m),  $x_2$ )
else if  $m = 0$  then
    return ( $x_1$ , mots_gaps(n))
else if  $n = 1$  then
    return align_lettre_mot( $x_1[0], x_2$ )11
else if  $m = 1$  then
    (a,b) = align_lettre_mot( $x_2[0], x_1$ )
    return (b,a)
else
     $j = \text{coupure}(x_1, x_2)$ 
     $i = \frac{n}{2}$ 
    return SOL_2( $x_1[0;i], x_2[0;j]$ ) · SOL_2( $x_1[i+1;n], x_2[j+1;m]$ )12
end if

```

---

#### Q.3.4.5 Complexité spatiale de coupure

La complexité spatiale de la fonction **coupure** est en  $O(m)$  car on utilise le tableau **Dist** rempli avec **Dist\_2**, qui a une complexité spatiale en  $O(m)$  ainsi que le tableau **I** de taille  $2 \times m$ , car on a besoin, à chaque étape, uniquement des lignes  $i$  et  $i - 1$  pour calculer l'indice à laquelle le chemin coupe la ligne  $i^*$ .

#### Q.3.4.6 Complexité spatiale de SOL\_2

SOL\_2 va créer deux chaînes de caractère dont la taille est en  $O(n + m)$ <sup>13</sup> en concaténant des plus petites. Puisque le tableau contenant les distances dans **coupure**<sup>14</sup> est réécrit à chaque étape, on peut optimiser notre complexité spatiale en créant ce tableau de taille en  $O(m)$  en amont de la fonction **Sol\_2** afin qu'il ne soit pas inutilement créé plusieurs fois. On utilisera alors en plus des deux chaînes en  $O(n)$  et  $O(m)$  un tableau de taille en  $O(m)$ .

La complexité spatiale de SOL\_2 sera donc en  $O(m + m + n + m + n + m) = O(n + m)$ .

---

<sup>13</sup>Comme monté en 1.1.2

<sup>14</sup>De même pour le tableau **I** de **coupure**

### Q.3.4.7 Complexité temporelle de coupure

$$\begin{aligned}
\sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} \sum_{j=0}^m 1 + \sum_{\lfloor \frac{n}{2} \rfloor}^n \sum_{j=0}^m i \times j &= \frac{n}{2} \times m + \sum_{\lfloor \frac{n}{2} \rfloor}^n i \times \sum_{j=0}^m j \\
&= \frac{n}{2} \times m + \sum_{\lfloor \frac{n}{2} \rfloor}^n i \frac{m(m+1)}{2} \\
&= \frac{n}{2} \times m + \frac{m(m+1)}{2} \times \sum_{\lfloor \frac{n}{2} \rfloor}^n i \\
&= \frac{n}{2} \times m + \frac{m(m+1)}{2} \times \left( \sum_{i=0}^n i - \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} i \right) \\
&= \frac{n}{2} \times m + \frac{m(m+1)}{2} \times \left( \frac{n(n+1)}{2} - \frac{\frac{n}{2}(\frac{n}{2}+1)}{2} \right) \\
&\in O(n^2 \times m^2)
\end{aligned}$$

### Q.3.4.8 Pseudo code de la fonction coupure

---

#### Algorithm 7 coupure

---

**Require:**  $x_1, x_2$ , deux séquences à aligner, de taille respectivement  $n$  et  $m$

$i^* = \lfloor \frac{n}{2} \rfloor$

$I[2][m]$  le tableau contenant nos indices de coupure

$Dist[2][m]$  le tableau qui nous permet de retenir pour chaque étape le tableau modifié par  $Dist\_2$  (et qui n'est pas retourné)

$i$  et  $j$  deux entiers

**for**  $i$  in range  $[0;n[$  **do**

**for**  $j$  in range  $[0;m[$  **do**

**if**  $i < i^*$  **then**

$I[1][j] = 0$

**else if**  $i = i^*$  **then**

$I[1][j] = j$

**else**

$DIST\_2(x_{[0;i]}, y_{[0;j]}, Dist)$

$chemin\_pris = \min\{Dist[0][j], Dist[0][j-1], Dist[1][j-1]\}$

**if**  $chemin\_pris = Dist[0][j]$  **then**

$I[1][j] = I[0][j]$

**else if**  $chemin\_pris = Dist[0][j-1]$  **then**

$I[1][j] = I[0][j-1]$

**else if**  $chemin = Dist[1][j-1]$  **then**

$I[1][j] = I[1][j-1]$

**end if**

**end if**

$I[0] = I[1]$

**end for**

**end for**

**return**  $I[1][m]$

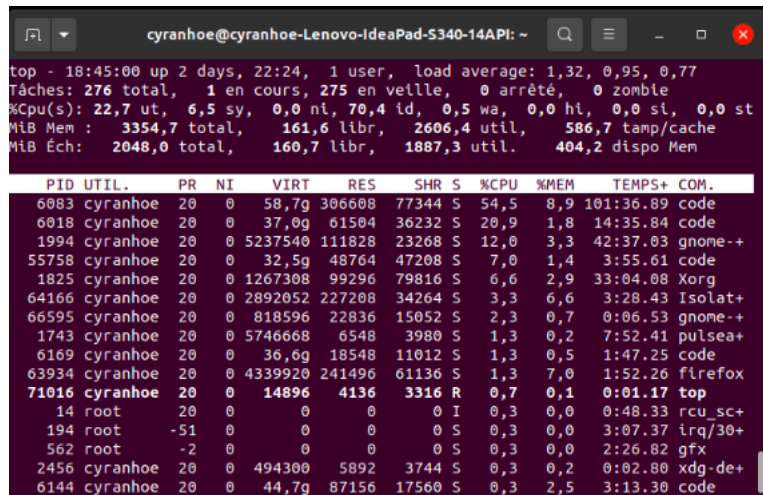
---

## Tâche D

*On n'a pas pu tracer la courbe de consommation temporelle car nos valeurs étaient soit nulles, soit incalculables sans que l'ordinateur ne plante*

### Consommation mémoire

Sur l'affichage de la console top, on voit que la consommation se fait en plusieurs fois du fait des différents appels récursifs. Mais elle ne semble pas excéder les 8,9%.



```
top - 18:45:00 up 2 days, 22:24, 1 user, load average: 1.32, 0.95, 0.77
Tâches: 276 total, 1 en cours, 275 en veille, 0 arrêté, 0 zombie
%Cpu(s): 22,7 ut, 6,5 sy, 0,0 ni, 70,4 id, 0,5 wa, 0,0 hi, 0,0 si, 0,0 st
MiB Mem : 3354,7 total, 161,6 libr, 2606,4 util, 586,7 temp/cache
MiB éch: 2048,0 total, 160,7 libr, 1887,3 util. 404,2 dispo Mem
```

PID	UTIL.	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TEMPS+	COM.
6083	cyranhoe	20	0	58,7g	306608	77344	S	54,5	8,9	101:36.89	code
6018	cyranhoe	20	0	37,0g	61504	36232	S	20,9	1,8	14:35.84	code
1994	cyranhoe	20	0	5237540	111828	23268	S	12,0	3,3	42:37.03	gnome-+
55758	cyranhoe	20	0	32,5g	48764	47208	S	7,0	1,4	3:55.61	code
1825	cyranhoe	20	0	1267308	99296	79816	S	6,6	2,9	33:04.08	Xorg
64166	cyranhoe	20	0	2892052	227208	34264	S	3,3	6,6	3:28.43	Isolat+
66595	cyranhoe	20	0	818596	22836	15052	S	2,3	0,7	0:06.53	gnome-+
1743	cyranhoe	20	0	5746668	6548	3980	S	1,3	0,2	7:52.41	pulsea+
6169	cyranhoe	20	0	36,6g	18548	11012	S	1,3	0,5	1:47.25	code
63934	cyranhoe	20	0	4339920	241496	61136	S	1,3	7,0	1:52.26	firefox
71016	cyranhoe	20	0	14896	4136	3316	R	0,7	0,1	0:01.17	top
14	root	20	0	0	0	0	I	0,3	0,0	0:48.33	rcu_sc+
194	root	-51	0	0	0	0	S	0,3	0,0	3:07.37	irq/30+
562	root	-2	0	0	0	0	S	0,3	0,0	2:26.82	gfx
2456	cyranhoe	20	0	494300	5892	3744	S	0,3	0,2	0:02.80	xdg-de+
6144	cyranhoe	20	0	44,7g	87156	17560	S	0,3	2,5	3:13.30	code

Figure 3.5: État de la mémoire de l'ordinateur après le lancement de `test_tache_D`

### Q.3.4.9 A-t-on perdu en complexité temporelle en améliorant la complexité spatiale ?

Oui, on a perdu en complexité temporelle en améliorant la complexité spatiale. La complexité temporelle de `SOL_1` était en  $O(n \times m)$ , celle de `SOL_2` est bien supérieure car elle utilise à chaque appel récursif `Dist_2` qui est en  $O(n \times m)$ .

# 4

## Une extension : l'alignement local de séquences (Bonus)

### Exercice 4.1 Approche théorique de cette extension

#### Q.4.1.1 Distance d'édition pour des valeurs de $x$ très grandes par rapport aux valeurs de $y$

On a vérifié cela expérimentalement en créant une instance d'adn `Instance_x_plus_grand_que_y.adn`, où  $n = 10000$  et  $m = 10$  à l'aide de `Dist_2`. L'algorithme nous a retourné une distance d'édition de

$$\begin{aligned} 19980 &= 10000 - 2 \times 10 \\ &= (|x| - |y|) \times c_{del} \end{aligned}$$

Ce qui est cohérent : la probabilité de trouver tous les nucléotides de  $y$  dans  $x$  dans l'ordre (avec des gaps entre-temps) est très forte compte tenu de la taille de cette dernière.

#### Q.4.1.2 Identifier les facteurs pertinents à aligner avec des gaps nuls

Avec la méthode proposée par l'énoncé, le coût d'un plus long alignement correspondrait uniquement aux substitutions, insertions, et suppressions qui se feraient "à l'intérieur" de l'alignement. Cela nous permet alors d'uniquement regarder ce qui se passe localement en forçant l'alignement à ne pas faire commencer l'apparition de caractères de  $y$  trop tôt, c'est à dire où il n'y aurait aucune correspondance vraiment exploitable de plusieurs nucléotides à la suite représentant un gène à étudier.

#### Q.4.1.3 Idée de méthode par scores d'alignements avec une complexité spatiale en $O(n \times m)$

Comme vu dans la partie précédente, la complexité spatiale de `SOL_1` est en  $O(n \times m)$ , on pourrait alors se baser sur cette méthode en modifiant les valeurs des coûts pour qu'ils correspondent à ceux proposés pour cette solution ainsi que le choix fait pour remonter dans le tableau.

Cependant, on ne pourrait pas améliorer la complexité spatiale comme fait avec `SOL_2` avec une méthode diviser pour mieux régner puisqu'il faudrait alors découper nos séquences, ce qui favorise la dispersion des valeurs dans nos alignements. De plus, cela se suivrait d'une perte du point de vue de la complexité temporelle.