Algorithmique LU3IN003

Rapport

Alignement de séquences

Préparé par :
Rami BENELMIR
Erisa KOHANSAL

Année universitaire
2022/2023

Table des matières

1 Le problème d'alignement de séquences

1.1 Question 1

Soit $(\bar{x}, \bar{y})$ est un alignement de (x, y) et $(\bar{u}, \bar{v})$ est un alignement de (u, v).
On veut démontrer que $(\bar{x}.\bar{u}, \bar{y}.\bar{v})$ est un alignement de $(x.u, y.v)$:
$I : |\bar{x}.\bar{u}| = |\bar{x}| + |\bar{u}| = |\bar{y}| + |\bar{v}| = |\bar{y}.\bar{v}|$
D'après les informations données, on sait que $\pi(\bar{x}) = \pi(x)$ et que $\pi(\bar{u}) = \pi(u)$. Or :
$II : \pi(\bar{x}.\bar{u}) = \pi(\bar{x}).\pi(\bar{u}) = x.u$
De même : $\pi(\bar{y}.\bar{v}) = \pi(\bar{y}).\pi(\bar{v}) = y.v$
$\forall i \in [1, \ldots, |\bar{x}|], \bar{x}_i \neq \_, \bar{y}_i \neq \_$ et $\forall j \in [1, \ldots, |\bar{u}|], \bar{u}_j \neq \_, \bar{v}_j \neq \_$ on a :
$III : \forall i + j = k \in [1, \ldots, |\bar{x}.\bar{u}|], (\bar{x}, \bar{u})_k \neq \_, (\bar{y}, \bar{v})_k \neq \_$
Conclusion : D'après I, II et III, cette propriété est vérifiée.

1.2 Question 2

La longueur maximale d'un alignement est alors majorée par $n + m$. Pour cela, on pourrait utiliser le caractères _ dans le but de créer un alignement de (x, y).
C'est le cas de quatrième exemple dans la partie 2.2 :
$\sum = A, T, G, C$ $x = ATTGTA$ et $y = ATCTTA$.
L'alignement de longueur maximale de (x, y) correspond à :
$\bar{x} = \_\_\_\_\_\_ATTGTA$
$\bar{y} = ATCTTA\_\_\_\_\_\_$
Donc on peut avoir jusqu'à $n + m$ de longueur maximum.
Conclusion : la longueur maximale d'un alignement de (x, y) est majorée par $n+m$

2 Algorithmes pour l'alignement de séquences

2.1 Question 3

On cherche le nombre de combinaisons possible de placer les gaps dans l'alignement de $(x, x+k)$. x est un mot de longueur n ($|x| = n$) or en ajoutant k gaps à x on obtient $\bar{x}$, c'est-à-dire : $|\bar{x}| = |x| + k = n + k$
Conclusion : $C_{\bar{x}+k}^{|x|} = C_{n+k}^{n}$ nous donne le nombre d'alignement de $(x, x+k)$ possible.

2.2 Question 4

$|x| = n$ et une fois qu'on ajoute k gaps à x, on aura $|\bar{x}| = |x| + k = n + k$
On peut obtenir le nombre de positionnement possible pour les k gaps à l'aide d'un coefficient binomial. Puisque qu'on n'a pas besoin de prendre en compte l'ordre des éléments, on utilise détermine le nombre de combinaison possibles : C_{n+k}^{k}
Une des conditions d'alignement est que les deux mots alignés n'aient pas de gap à la même position. D'après cette condition et d'après le fait que $n \geq m$ alors pour satisfaire ces deux conditions, k doit être plus petit que m ($k \leq m$).
Néanmoins, le nombre de gap dans y est égal à la différence de taille de x et y plus le k nombre de gaps ajoutés à x (c'est-à-dire $n - m + k$, qui est toujours positif) : C_{n}^{n-m+k}
En prenant en compte toutes ces contraintes, on aura : $C_{n+k}^{k}.C_{n}^{n-m+k}$
Conclusion : $C_{n+k}^{k}.C_{n}^{n-m+k}$ Sera le nombre total d'alignements possibles. Ainsi pour $|x| = 15$ et $|y| = 10$ on a :

$\sum_{k=0}^{10} \frac{(k+15)!15!}{15!k!(10-k)!(5+k)!} = 298199265$ alignements possibles.

2.3 Question 5

La complexité temporelle, c'est le calcul de tous les alignements possibles, pour apres choisir celui qui à la distance la plus minimal.
Pour le calulcul de de tout les alignements possibles ce fait en : $\sum_{0}^{m} C_{n+k}^{k}.C_{n}^{n-m+k}$.
Et pour enumerer tous les alignements pour trouver l'alignements du cout minimal le cout sera négligé, parce que c'est trop petit « par rapport au cout ddu calcul des alignement.

Conclusion : On est sur une complexité temporelle de $O(\sum_{0}^{m} C_{n+k}^{k}.C_{n}^{n-m+k})$, et c'est trés couteux.

2.4 Question 6

Pour trouver l'alignement minimal on doit avoir un variable qui stock la distance, et un tableau de taille max d'alignement qui est n+m.
Conclusion : On est sur une complexité spatialle de O(n+m)

2.5 Tâche A

On a eu les bons résultats lors d'implémentation de "DIST_NAIF" sur les trois instances "Inst_0000010_44.adn", "Inst_0000010_7.adn" et "Inst_0000010_8.adn".

```
print("Instances_genome\Inst_0000010_44.adn")
_, _, x, y = readFile("Instances_genome\Inst_0000010_44.adn")
print("distance d'édition : ", DIST_NAIF(x, y))

print("Instances_genome\Inst_0000010_7.adn")
_, _, x, y = readFile("Instances_genome\Inst_0000010_7.adn")
print("distance d'édition : ", DIST_NAIF(x, y))

print("Instances_genome\Inst_0000010_8.adn")
_, _, x, y = readFile("Instances_genome\Inst_0000010_8.adn")
print("distance d'édition : ", DIST_NAIF(x, y))
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER

Instances_genome\Inst_0000010_44.adn
distance d'édition :  10
Instances_genome\Inst_0000010_7.adn
distance d'édition :  8
Instances_genome\Inst_0000010_8.adn
distance d'édition :  2
```

FIGURE 1 – Les résultats rendus par "DIST_NAIF" qui est exécutée sur les instances "Inst_0000010_44.adn", "Inst_0000010_7.adn" et "Inst_0000010_8.adn"

Pour déterminer l'instance pour laquelle le temps d'exécution de la fonction "DIST_NAIF" va dépasser une minute, on lance la fonction "trace_courbe_tacheA" situé dans le fichier "traceFunc.py". Cette fonction se termine lorsque le temps d'xécution d'une instance dépasse une minute, qui est l'instance "Inst_0000012_56.adn".

```
tailleA, tempsA = trace_courbe_tacheA("Instances_genome\\")

PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER
l'exécution de  Instances_genome\Inst_0000012_56.adn  a pris  83.41903805732727  seconds qui est plus d'une minute
```

FIGURE 2 – L'exécution de la fonction "DIST_NAIF" prend plus d'une minute pour l'instance "Inst_0000012_56.adn"

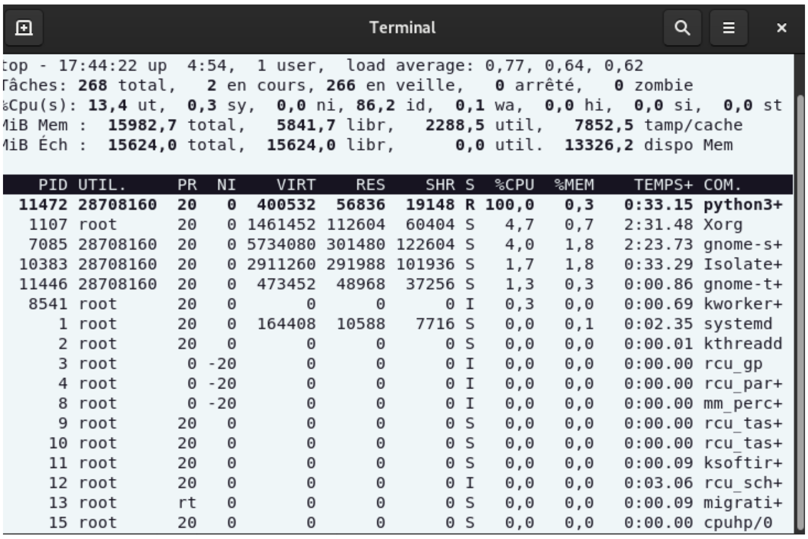

FIGURE 3 – Consommation de mémoire par la fonction "DIST_NAIF"

Sur la figure 2, la première ligne correspond à l'exécution de "DIST_NAIF" sur l'instance "Inst_0000012_56.adn". Cette fonction consomme 0.3% de la mémoire. Vu que la taille de cette instance n'est en effet pas si grande, cette consommation de mémoire est élevée.

3 Programmation dynamique

3.1 Question 7

Si $\bar{u}_l = \_$, d'après les conditions d'alignement on ne peut pas avoir un gap sur la même position dans les deux mots ($\bar{u}_l$ étant le l-ième élément de $\bar{u}$). Or cela implique que $\bar{v}_l = y_j$ (y_j est le j-ième élément de y). De même pour le cas où on a $\bar{v}_l = \_$, alors $\bar{u}_l = x_i$ (x_i est le i-ième élément de x). Maintenant si $\bar{u}_l \neq \_$ et $\bar{v}_l \neq \_$, cela veut dire que $\bar{u}_l = x_i$ et $\bar{v}_l = y_j$.

3.2 Question 8

$$C(\bar{u},\bar{v}) = C(\bar{u}_{[1,\ldots,l-1]}, \bar{v}_{[1,\ldots,l-1]}) + \begin{cases} C_{ins}, & \text{si } \bar{u}_l = 0 \\ C_{del}, & \text{si } \bar{v}_l = 0 \end{cases}$$

3.3 Question 9

On sait que la distance est calculée récursivement, on ajoute à chaque fois le coût minimal des opérations. D'après les deux questions précédentes, on peut conclure :

$$D(i,j) = min \begin{cases} D(i-1,j) + C_{ins} \\ D(i,j-1) + C_{del} \\ D(i-1,j-1) + C_{sub}(x_i,y_j) \end{cases}$$

3.4 Question 10

$D(0,0) = d(\emptyset,\emptyset) = 0$ c'est la distance entre deux mots vides.

3.5 Question 11

— $D(0,j)$ consiste en l'ajout de gap dans x. C'est-à-dire qu'on va répéter l'opération d'insertion j fois (un alignement de longueur j) :

$$\begin{aligned} D(0,j) &= D(0,j-1) + C_{ins} \\ &= [D(0,j-2) + C_{ins}] + C_{ins} = D(0,j-2) + 2 \times C_{ins} \\ &= [D(0,j-3) + C_{ins}] + 2 \times C_{ins} = D(0,j-3) + 3 \times C_{ins} \\ &= \ldots \\ &= D(0,j-j) + j \times C_{ins} = D(0,0) + j \times C_{ins} = j \times C_{ins} \end{aligned}$$

D'après la question précédente $D(0,0) = 0$.
Alors on a : $D(0,j) = j \times C_{ins}$

— $D(i,0)$ consiste en l'ajout de gap dans y. C'est-à-dire qu'on va répéter l'opération de suppression i fois de suite (un alignement de longueur i) :

$$\begin{aligned} D(i,0) &= D(i-1,0) + C_{del} \\ &= [D(i_2,0) + C_{del}] + C_{del} = D(i-2,0) + 2 \times C_{del} \\ &= [D(i-3,0) + C_{del}] + 2 \times C_{del} = D(i-3,0) + 3 \times C_{del} \\ &= \dots \\ &= D(i-i,0) + i \times C_{del} = D(0,0) + i \times C_{del} = i \times C_{del} \end{aligned}$$

D'après la question précédente $D(0,0) = 0$.
Alors on a : $D(i,0) = i \times C_{del}$

Conclusion : $D(0,j) = j \times C_{ins}$ et $D(i,0) = i \times C_{del}$

3.6 Question 12

Algorithm 1 DIST_1

Require: Deux mots x , y
Ensure: Un tableau T

$n = longueur(x)$
$m = longueur(y)$
$T[n][m] = 0$
for $j = [1 \dots n]$ **do**
    T[i][0] ← i * Cdel
end for
for $j = [1 \dots m]$ **do**
    T[0][j] ← i * Cins
end for
for $i = [1 \dots n]$ **do**
    for $j = [1 \dots m]$ **do**
        $T[i][j] \leftarrow min(T[i-1][j-1] + Csub(x[i-1], y[j-1])); T[i][j-1] + Cins; T[i-1][j] + Cdel)$
    end for
end for
return T

3.7 Question 13

La complexité spatialle de DIST_1 est en $\theta(n \times m)$ car on stocke le coût dans chaque case de notre matrice T.

3.8 Question 14

La complexité temporelle de DIST_1 est en $\theta(n \times m)$ car on a deux boucles imbriquées allant de 0 à n-1 et de 0 à m-1. Toutes les autres opérations sont des opérations élémentaires.

3.9 Question 15

On cherche à démontrer que :
Si $i > 0$ *et* $D(i,j) = D(i-1,j) + C_{del},$ *alors* $\forall(\bar{s},\bar{t}) \in Al^*(i-1,j),$
$(\bar{s}.x_i, \bar{t}.\_) \in Al^*(i,j)$
Commençons par

$$\begin{aligned} D(i,j) &= D(i-1,j) + C_{del} \\ &= d(x_{[1,\ldots,i]}, y_{[1,\ldots,j]}) + C_{del} \\ &= C(\bar{s}.x_i, \bar{t}.\_) \end{aligned}$$

Or $C(\bar{s}.x_i, \bar{t}.\_) \ \in Al^*(i,j)$

3.10 Question 16

Algorithm 2 SOL_1

Require: Deux mots x , y et un tableau T
Ensure: Un tableau T

```
i ← longueur(x)
j ← longueur(y)
x̄ = [ ]
ȳ = [ ]
while i > 0 and j > 0 do
    if i>0 and j>0 and T(i,j)= T(i-1,j-1)+Csub(x[i-1],y[j-1]) then
        x̄ = x_i.x̄
        ȳ = y_j.ȳ
        i − −
        j − −
    else
        if i>0 and T(i,j)=T[i-1][j]+Cdel then
            x̄ = x_i.x̄
            ȳ = −.ȳ
            i − −
        else
            if j>0 and T(i,j)=T[i][j-1]+Cins then
                x̄ = −.x̄
                ȳ = y_j.ȳ
                j − −
            end if
        end if
    end if
end while
return (x̄, ȳ)
```

3.11 Question 17

Notre probléme ALI se résolue avec deux fonctions :
— DIST_1 se calcule en O(nm).
— SOL_1 se calcule en O(n+m).
Conclusion : On a une complexité temporelle de O(nm).

3.12 Question 18

Notre probléme ALI se résolue avec deux fonctions :

- DIST_1 qui alloue un tableau de taille n*m, ce qui une complexité spatialle de O(nm).
- SOL_1 qui alloue deux tableau de taille n et m, ce qui donne une complexité spatialle de O(n+m).

  Conclusion : On a une complexité spatialle de O(nm).

3.13 Tâche B

On utilise la fonction "trace_courbe_tacheB" qu'on a implémentée dans le fichier "traceFunc.py" afin d'obtenir la courbe de consommation de temps CPU en fonction de la taille $|x|$: La première ligne de la figure 3 correspond à la

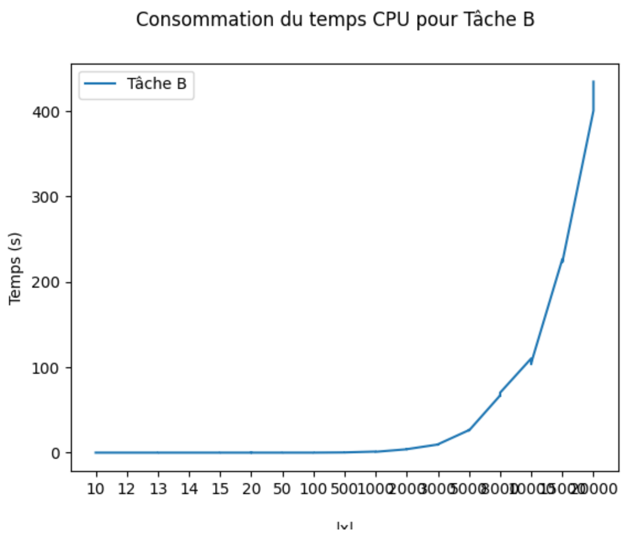

FIGURE 4

consommation mémoire de "PROG_DYN" exécutée sur une instance de grande taille "Inst_0010000_7.adn". Cette consommation est très importante, vu que ça consomme 4.6% de la mémoire.

3.14 Question 19

Car à chaque tour de boucle la valeur minimum de $D(i,j)$ est calculée à partir de $D(i-1,j-1), D(i-1,j), D(i,j-1)$ or on a besoin que de ces deux lignes i et $i-1$. Alors, on pourra utiliser une matrice à deux lignes afin d'améliorer la complexité spatiale.

```
x-terminal-emulator
top - 19:25:22 up 1 day,  8:16,  3 users,  load average: 1,34, 1,50, 1,19
Tâches: 305 total,   2 en cours, 303 en veille,   0 arrêté,   0 zombie
%Cpu(s): 15,3 ut,  1,1 sy,  0,0 ni, 83,3 id,  0,3 wa,  0,0 hi,  0,0 si,  0,0 st
MiB Mem :  15892,0 total,   2912,2 libr,   3770,5 util,   9209,3 tamp/cache
MiB Éch :  15625,0 total,  15625,0 libr.,      0,0 util.  11686,4 dispo Mem

    PID UTIL.      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TEMPS+ COM.
  67017 28708160   20   0 1096044 751632  19152 R  99,7   4,6    0:44.08 python3.9
  52030 28708160   20   0 3244040 563776 185560 S  14,6   3,5   12:36.31 firefox-esr
  32567 root       20   0 6557036 135096  52072 S   8,3   0,8    2:35.57 Xorg
  52731 28708160   20   0 9298496 359420 119056 S   6,0   2,2    0:36.94 Isolated Web Co
  26036 root      -51   0       0      0      0 S   2,0   0,0    0:52.08 irq/130-nvidia
  51787 28708160   20   0  323792  90296  52204 S   1,7   0,6    0:14.60 xfwm4
  67142 28708160   20   0  391108  40596  32860 S   1,3   0,2    0:00.52 gnome-screensho
  67005 28614429   30  10   14584   4756   4212 S   0,7   0,0    0:00.13 deco
```

FIGURE 5 – Consommation mémoire de la fonction "PROG_DYN" exécutée sur l'instance "Inst_0010000_7.adn"

3.15 Question 20

Algorithm 3 DIST_2

Require: Deux mots x , y
Ensure: Distance entre x et y

$n = longueur(x)$
$m = longueur(y)$
$T[2][m] = 0$
for $j = [1 \ldots m]$ **do**
    T[0][j] ← i * Cins
end for
for $i = [0 \ldots n]$ **do**
    for $j = [0 \ldots m]$ **do**
        $T[1][j] \leftarrow min(T[0][j-1] + Csub(x[i-1], y[j-1])); T[1][j-1] + Cins; T[0][j] + Cdel)$
    end for
    for $j = [0 \ldots m]$ **do**
        $T[0][j] \leftarrow T[1][j]$
    end for
end for
return T

3.16 Tâche C

Pour le traçage de la courbe de consommation de temps CPU en fonction de la taille $|x|$, on a implémenté la fonction "trace_courbe_DIST_2" dans le fichier "traceFunc.py".

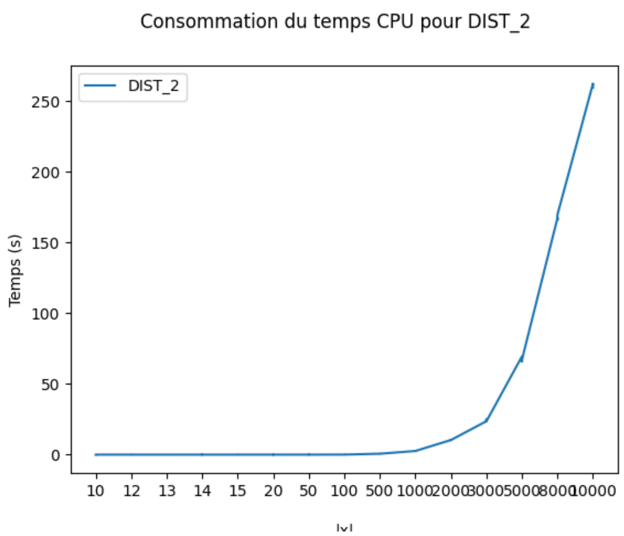

FIGURE 6

On peut alors comparer les deux fonctions "DIST_1" et "DIST_2" :

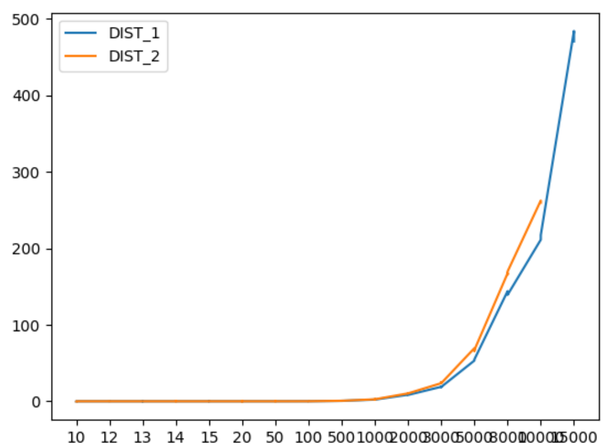

FIGURE 7

D'après la figure 7, on peut voir que la courbe de "DIST_2" monte moins vite par rapport à celle de "DIST_1". Cela veut dire qu'on a réussi à améliorer la complexité

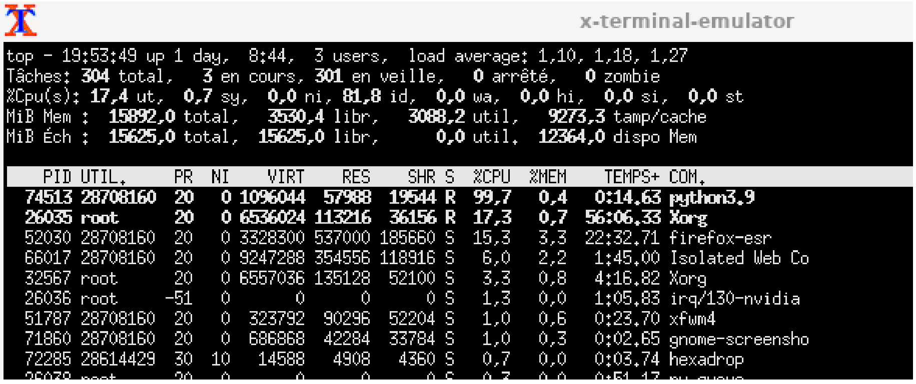

FIGURE 8 – Consommation de la mémoire par la fonction "DIST_2" pour une instance d'une grande taille, ici on a utilisé "Inst_0010000_7.adn"

temporelle de "DIST_2" par rapport à celle de "DIST_1".
La première ligne sur la figure 8 correspond aux détails de l'exécution de la fonction "DIST_2" sur l'instance "Inst_0010000_7.adn". Cette exécution occupe 0.4% de la mémoire. En comparant ce résultat avec celui de "DIST_1" dans les questions précédentes, on conclut que la consommation de mémoire a diminué de $4.6\% - 0.4\% = 4.4\%$.
Consclusion : En conclusion, "DIST_2" a une complexité spatialle (resp. une complexité temporelle) qui est plus petite que celle de "DIST_1".

3.17 Question 21

Algorithm 4 mot_gaps

Require: Une entier k
Ensure: Un mot de k gaps

mot = [ ]
$mot \leftarrow' -' * k$
return (mot)

3.18 Question 22

Algorithm 5 align_lettre_mot

Require: Deux mots x , y
Ensure: Alignement de x et y
for $i = [0 \ldots longueur(y) - 1]$ **do**
  if x[i] == y[i] **then**
    Sortir de la boucle
  end if
end for
$x \leftarrow mot\_gaps(i) + x + mot\_gaps(longueur(y) - i - 1)$
return (x, y)

3.19 Question 23

On decoupe x et y au milieu : x_1 : BAL, x_2 : LON, y_1 : RO, y_2 : ND.

Alignement de (x_1, y_1) : (B , A, L) , (R , O, -) avec une distance de 13.

Alignement de (x_2, y_2) : (L , O, N, -) , (- , -, N, D) avec une distance de 9.

Alignement de $(x_1.x_2, y_1.y_2)$: (B , A, L ,L , O, N, -) , (R , O, - ,- , -, N, D) avec une distance de 22.

Alors l'alignement optimat est : (B , A, L ,L , O, N, -) , (R , - ,- , -, O, N, D) qui a une distance de 17.

Consclusion : par absurde (B , A, L ,L , O, N, -) , (R , O, - ,- , -, N, D) n'est pas un alignement optimal.

3.20 Question 24

Algorithm 6 SOL_2_REC

Require: Deux mots x , y
Ensure: Distance entre x et y

$n = longueur(x)$
$m = longueur(y)$
if $n == 0$ **then**
    return $(mot\_gaps(m), y)$
end if
if $m == 0$ **then**
    return $(x, mot\_gaps(n))$
end if
if $n = 1$ **then**
    return $align\_lettre\_mot(x, y)$
else
    index_i = n div 2
    index_j = coupure(x,y)
    $(x1, y1) \leftarrow SOL\_REC\_2(x_{[0...index\_i]}, y_{[0...index\_j]})$
    $(x2, y2) \leftarrow SOL\_REC\_2(x_{[index\_i+1...n]}, y_{[index\_j+1...m]})$
end if
return $(x1 + x2, y1 + y2)$

3.21 Question 25

Algorithm 7 Coupure

Require: Deux mot x et y
Ensure: L'indice j*

```
T : Une liste de deux case pour les distances
c_l i : Une liste de deux cases pour stocker la colonne sur la ligne i
for j = [0 ... longueur(y)]  do
    T[0].append( j * Cins )
    T[1].append( -1 )
    c_l i [0].append( j )
    c_l i [0].append( -1 )
end for
for i = [1... longueur(n) div 2] do
    T[1][0] ← i * Cdel
    for j = [1 ... longueur(y)] do
        T[1][j] ←min (T[0][j-1]+Csub(x[i-1],y[j-1])) ;T[1][j-1]+Cins ;T[0][j]+Cdel )
    end for
end for
for i = [longueur (n) div 2 + 1... longueur (n)  do
    T[1][0] ← i * Cdel
    for j = [1 ... longueur(y)] do
        T[1][j] ←min (T[0][j-1]+Csub(x[i-1],y[j-1]) ;T[1][j-1] +Cins ;T[0][j]+Cdel )
        If T[1][j]=T[1][j-1] +Cins then
        c_l i[1][j] ← c_l i [1][j-1]
        If T[1][j]=T[0][j]+Cdel then
        c_l i[1][j] ← c_l i [0][j]
        If T[1][j]=T[0][j-1]+Csub(x[i-1],y[j-1]) then
        c_l i[1][j] ← c_l i [0][j-1]
    end for
    c_l i[0] ← c_l i [1]
end for
return c_l i[0][m]
```

3.22 Question 26

La complexité spatialle de notre fonction coupure est de O(m), étant donné qu'on deux tableau (liste pour notre cas) de deux cases de taille m.

3.23 Question 27

On sait que la complexité de coupure et en O(m), on a : n>m donc : O(n). Et a chaque appel récursif on dévise les deux mots x et y en deux et on lance deux appels. En utilisant le théoreme maitre on trouve : T(n) = 2 T($\frac{n}{2}$) +O(n).
SOl_2_REC est de complexité : O(nlog(n)).

3.24 Question 28

Etant donné qu'on doit faire n*m itérations pour trouver la bonne coupure. la complexité temporelle de coupure est de O(nm).

3.25 Tache D

La fonction "trace_courbe_tacheD" dans le fichier "traceFunc.py" nous permet de tracer la courbe de consommation de temps CPU de la tâche D en fonction de $|x|$. La première ligne de la figure ci-dessus contient les détails de l'exécution de la

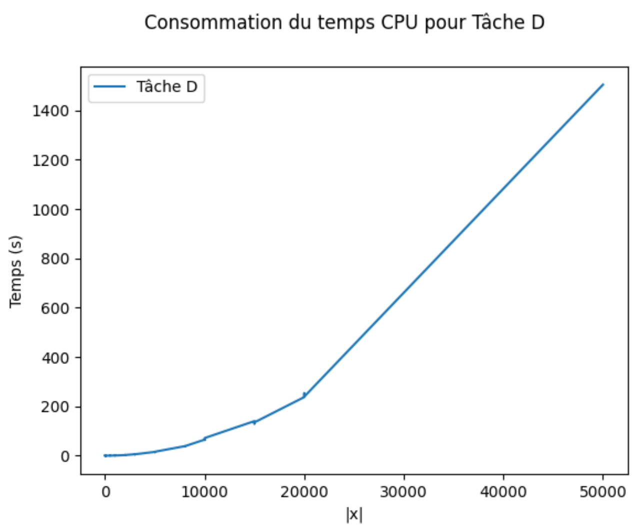

FIGURE 9

fonction "coupure" pour une grande instance, "Inst_0010000_7.adn". On a obtenu cette figure grâce à la commande "top" de Linux. La consommation mémoire étant égale à 0.4%, cette valeur n'est pas une grande consommation par rapport à la taille de notre instance.

```
x-terminal-emulator
top - 20:06:37 up 1 day,  8:57,  4 users,  load average: 2,13, 1,76, 1,53
Tâches: 316 total,   4 en cours, 312 en veille,   0 arrêté,   0 zombie
%Cpu(s): 24,3 ut,  2,3 sy,  1,2 ni, 71,8 id,  0,4 wa,  0,0 hi,  0,0 si,  0,0 st
MiB Mem :  15892,0 total,   2786,4 libr,   3776,1 util,   9329,5 tamp/cache
MiB Éch :  15625,0 total,  15625,0 libr,      0,0 util.  11614,8 dispo Mem

  PID UTIL.      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TEMPS+ COM.
78203 28708160   20   0 1097004  58712  19016 R 100,3   0,4   0:10.34 python3.9
26035 root       20   0 6536024 113216  36156 R  69,7   0,7  57:44.20 Xorg
52030 28708160   20   0 3657868 748828 211532 S  16,0   4,6  27:58.19 firefox-esr
77637 28614429   30  10   24796  15108   4496 S  15,0   0,1   0:18.79 metaballs
32567 root       20   0 6574604 152320  67564 S   9,7   0,9   5:53.60 Xorg
75250 28708160   20   0 9221952 360080 119240 S   6,3   2,2   0:20.85 Isolated Web Co
26036 root      -51   0       0      0      0 S   2,0   0,0   1:14.83 irq/130-nvidia
```

FIGURE 10 – La consommation mémoire de la fonction "coupure"

3.26 Question 29

Oui, en voulant améliorer la complexité spatiale on a perdu en complexité temporelle. On remarque bien que la complexité temporelle de SOL_2_REC est bien supérieur de celle de SOL_1.

4 L'alignement local de séquences (BONUS)

4.1 Question 30

En s'inspirant des instances fournies, on a créée trois nouvelles instances "Instance_0010000_5.adn", "Instance_0015000_9.adn" et "Instance_0020000_12.adn" situés dans le dossier "Question30". Dans le fichier "main.py", on a exécuté la fonction "DIST_2" sur ces trois instances et on a obtenu les résultats ci-dessous :

```
_, _, x, y = readFile("Question30\\Instance_0010000_5.adn")
print(DIST_2(x,y))
_, _, x, y = readFile("Question30\\Instance_0015000_9.adn")
print(DIST_2(x,y))
_, _, x, y = readFile("Question30\\Instance_0020000_12.adn")
print(DIST_2(x,y))

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER

19990
29982
39976
```

FIGURE 11

Ces résultats sont cohérents avec notre hypothèse :

$$|x| = 10000, |y| = 5, (|x| - |y|) \times C_{del} = 19990$$
$$|x| = 15000, |y| = 9, (|x| - |y|) \times C_{del} = 29982$$
$$|x| = 20000, |y| = 12, (|x| - |y|) \times C_{del} = 39976$$

Conclusion : On conclut que le coût d'un alignement global (x, y) quand $|x| >> |y|$ relativement à $|\sum|$ vaut $|x| - |y| \times C_{del}$

4.2 Question 31

Cela semble être une bonne idée pour les alignements petits, mais dans les pire des cas, c'est-à-dire quand la taille d'alignement et $n + m$ ($|x| = n, |y| = m$), dans ce cas là on doit d'abord parcourir tout l'alignement afin d'enlever les gaps, donc une boucle en $n + m$. Puis, une deuxième boucle $n + m$.
On aura une compléxité de $O(n + m)^2$. ce qu'est couteux.