Sorbonne Université

Mini-projet : Alignement de séquences

LU3IN003 - Algorithmique

Par Amel CHETTI et Inas KACI

4 décembre 2019

Alignement de séquences

Amel CHETTI, Inas KACI

October 2019

Table des matières

1 Introduction

Dans le cadre du mini-projet : Alignement de séquences de l'UE d'algorithmique de L3 Informatique à Sorbonne Université, nous présentons dans ce rapport nos réponses aux questions théoriques et les études expérimentales que nous avons effectué.

2 Le problème d'alignement des séquences :

2.0.1 Réponse 1 :

On suppose $(\overline{x}, \overline{y})$ est un alignement de (x, y) et $(\overline{u}, \overline{v})$ est un alignement de (u, v)
La fonction π est un morphisme de conoïdes libres alors :

$$\pi(\overline{x}.\overline{v}) = \pi(\overline{x}).\pi(\overline{v}) = x.v \tag{1}$$

On pose $n = |\overline{x}| = |\overline{y}|$ et $m = |\overline{u}| = |\overline{v}$ alors étant donné que la longueur de deux mots concaténés correspond à la somme des longueurs de chacun des mots : $|\overline{x}.\overline{u}| = |\overline{x}| + |\overline{u}| = n + m$
$|\overline{y}.\overline{v}| = |\overline{y}| + |\overline{v}| = n + m$
donc

$$|\overline{x}.\overline{u}| = |\overline{y}.\overline{v}| \tag{2}$$

si$a = x.u$ est le mot $a_1...a_{n+m} = x_1x_2...x_nu_1u_2...u_m$ et $b = y.v$ est le mot $b_1...b_{n+m} = y_1y_2...y_nv_1v_2...v_m$ alors on sait par hypothèse que :
$\forall i \in [1...n]$ $\overline{x_i} \neq -$ ou $\overline{y_i} \neq -$ alors $\forall i \in [1...n]$ $a_i \neq -$ ou $b_i \neq -$ $\forall i \in [1...m]$ $\overline{u_i} \neq -$ ou $\overline{v_i} \neq -$ alors $\forall i \in [n + 1...m]$ $a_i \neq -$ ou $b_i \neq -$
On déduit donc :

$$\forall i \in [1...n + m] \quad a_i \neq - \quad ou \quad b_i \neq - \tag{3}$$

D'après la définition 2.2, de (1), (2), (3) on déduit que $(\overline{x}.\overline{u}, \overline{y}.\overline{v})$ **est un alignement de** $(x.u, y.v)$

2.0.2 Réponse 2 :

$x \in \Sigma^*$ est un mot de longueur n , $y \in \Sigma^*$ est un mot de longueur m
$(\overline{x}, \overline{y})$ un alignement de (x, y), tel que $|\overline{x}| = |\overline{y}|$ et pour tout $i \in [1...|\overline{x}|]$, $\overline{x}_i \neq -$ ou $\overline{y}_i \neq -$
Montrons par l'absurde que $|\overline{x}| = |\overline{y}| \leq m + n$:

— On a suppose que $|\overline{x}| = |\overline{y}| > m + n$.
On a pour tout $i \in [1...|\overline{x}|]$, $\overline{x}_i \neq -$ ou $\overline{y}_i \neq -$.Alors la somme suivante : cardinal de l'ensemble $\{\overline{x}_i \neq -\}$+cardinal de l'ensemble $\{\overline{y}_i \neq -\}$, est égale à $|\overline{x}|$.

Or $|x|$ = cardinal de l'ensemble $\{\overline{x}_i \neq -\}$
$|y|$ = cardinal de l'ensemble $\{\overline{y}_i \neq -\}$
donc $|x| + |y| = |\overline{x}| > m + n \rightarrow$ Contradiction car $|x| + |y| = m + n$.
Donc $|\overline{x}| = |\overline{y}| \leq m + n$.

D'une autre manière : On suppose que k la longueur maximale de l'alignement $(\overline{x}, \overline{y})$ de (x,y). Alors $\forall i \in [1, ..., k]$ on a soit $x_i = -$ soit $y_i = -$. (D'après la définition)
Et comme : $|x|$=n et $\pi(\overline{y}) = y$, on peut avoir dans $\overline{y}$ au plus n gaps. $|\overline{y}|$=$|y| + n$= n+m.
Donc on a au plus m gaps dans x, c'est à dire qu'on peut au maximum avoir $\forall y_i \in \sum$, $x_i = -$ et vice versa, $\forall i \in [1, ..., m]$ si $y_i = -$ alors on a forcément $x_i \in \sum$, donc on peut avoir au plus n gaps dans y.
On déduit donc que la longueur maximale d'un alignement de (x, y) **est** $n + m$**.**

Exemple : Si on prend $x =$ "AGT", $y =$ "$TACG$"
$|x| = 3$ et $|y| = 4$
Soit un alignement qui suit : $\overline{x} =$ "$AGT - - - -$" et $\overline{y} =$ " $- - - TACG$"
Cet alignement est de longueur $7 = |x| + |y|$

Soit ($\overline{x}$,$\overline{y}$) un alignement de (x,y), $|x| = n$ et $|y| = m$.

3 Algorithmes pour l'alignement des séquences :

3.1 Méthode naïve par énumération :

Étant donné $x \in \Sigma^*$ un mot de longueur n, on note Nb le nombre de mots $\overline{x}$ obtenus en insérant à x exactement k gaps.
Nb correspond au nombre de sous ensembles possibles de longueur k (nombre de gaps) parmi un ensemble de longueur $(n + k)$ (le nombre des caractères de $\overline{x}$).
Donc

$$Nb = \binom{n+k}{k} = \frac{(n+k)!}{k!n!}$$

3.1.1 Réponse 4 :Le nombre d'alignements possibles

(x,y) deux mots de longueurs respectives n et m. On suppose que $n \geq m$.
Une fois ajoutés k gaps à x pour obtenir un mot $\overline{x} \in \Sigma^*$, il faut ajouter $k + (n - m)$ gaps à y.

Au total on a $\binom{k+n}{k+(n-m)}$ façons d'insérer $k + (n - m)$ gaps dans y (D'après la question 3).
Or un gap du mot $\overline{y} \in \Sigma^*$ainsi obtenu doit être placé à une position différente que celle d'un gap de $\overline{x}$.
Sachant que $\overline{x}$ possède k gaps, le nombre possible d'alignements est :

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

Cela revient à positionner $k+n-m$ gaps parmi n places.

Le nombre d'alignements possibles de (x, y) :

$$\sum_{k=0}^{m} \binom{n}{m-k} \binom{n+k}{k} = \sum_{k=0}^{m} \frac{(n+k)!}{(k+n-m)!(m-k)!k!}$$

tel que $k = |\overline{x}| - |x| = |\overline{x}| - n$, $|\overline{x}| \geq n$ et $|\overline{x}| \leq |x| + |y| = n + m$
Alors $0 \leq k \leq m$
.
Calcul sur machine du nombre d'alignements possibles de (x,y), à l'aide de l'expression ci-dessus avec $n = 15$ et $m = 10$:

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

3.1.2 Réponse 5 :Complexité temporelle

Considérons un algorithme naïf qui énumère tous les alignements possibles de deux mots x et y, de longueurs respectives n et m, en vue de trouver la distance d'édition entre ces deux mots.
Cherchons la complexité temporelle de cet algorithme.
Le nombre d'alignements possibles de (x,y) est

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

En mathématiques combinatoires, on a l'identité de Vandermonde :

$$\sum_{j=0}^{r} \binom{N}{j} \binom{M}{r-j} = \binom{N+M}{r}$$

On a également : $\sum_{k=0}^{10} \binom{n}{k} = \sum_{k=0}^{10} \binom{n}{k} 1^k 1^{n-k} = (1+1)^n = 2^n$
Comme $\binom{n+k}{k} > \binom{n}{k} \forall k \in [0, m]$, alors $\binom{n}{m-k}\binom{n+k}{k} > \binom{n}{m-k}\binom{n}{k} \forall k \in [0, m]$
et $\sum_{k=0}^{m} \binom{n}{m-k}\binom{n+k}{k} > \sum_{k=0}^{m} \binom{n}{m-k}\binom{n}{k}$

On pose : $N = n, M = n$ et $r = m$. On obtient donc

$$\sum_{k=0}^{m} \binom{n}{k} \binom{n}{m-k} = \sum_{k=0}^{r} \binom{N}{k} \binom{M}{r-k} = \binom{N+M}{r} = \binom{n+n}{m} = \binom{2n}{m}$$

Le nombre d'alignements possibles est alors minoré par $\binom{2n}{m}$.
Dans le cas où $m = n$, le nombre d'alignements possibles est minoré par $\binom{2n}{n}$.
Or en utilisant la formule de Stirling, on peut remplacer $n!$ par $\sqrt{2\pi n}(\frac{n}{e})^n$.

On peut donc expliciter le minorant :
$\binom{2n}{n} = \frac{2n!}{n!n!} = \frac{\sqrt{4\pi n}(\frac{2n}{e})^{2n}}{\sqrt{2\pi n}(\frac{n}{e})^{n}\sqrt{2\pi n}(\frac{n}{e})^{n}} = \frac{(2n)^{2n}}{n^{2n}\sqrt{\pi n}} = \frac{2^{2n}}{\sqrt{\pi n}}$

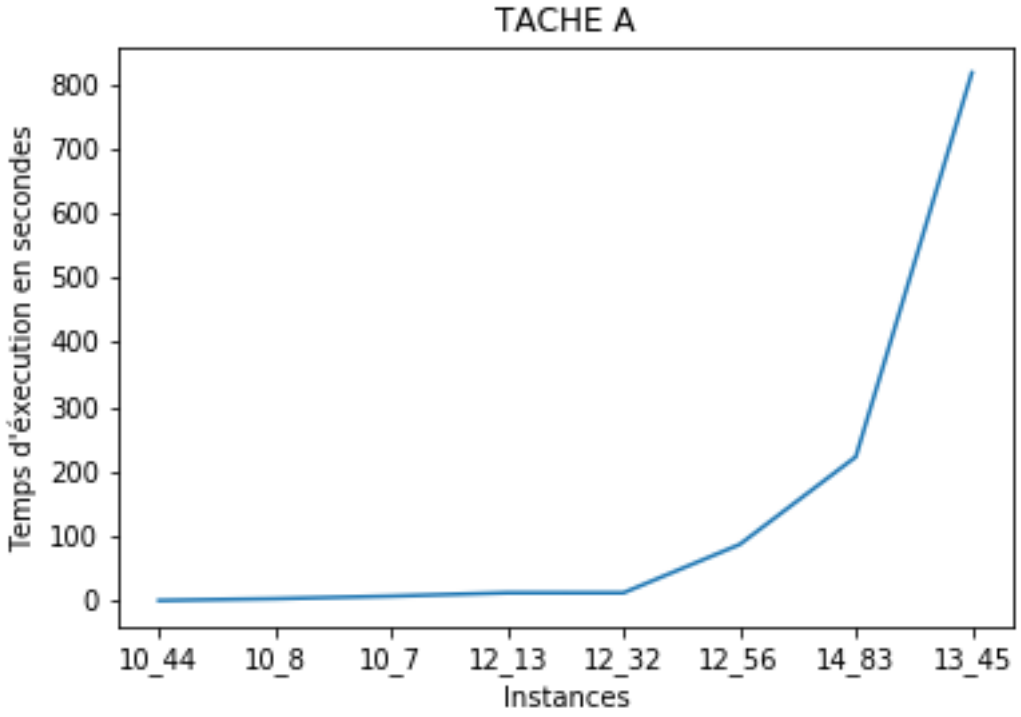
Le nombre d'alignements possibles d'un couple de mots de longueurs respectives n et m est de l'ordre de $O(2^n)$.
Les coûts de traitements effectués pour trouver un alignement de (x, y) sont ignorés du fait que le nombre possible de tous les alignements possibles est en $O(2^n)$ (exponentielle).Alors la complexité temporelle de cet algorithme naïf qui consisterait à énumérer tous les alignements de deux mots en vue de trouver la distance d'édition entre ces deux mots ou de trouver un alignement de coût minimal est la même, elle est de l'ordre de $O(2^n)$ $\times$ coût d'un alignement qui est équivalent àO(2^n) ,car la distance d'édition correspond au coût du meilleur alignement(de coût minimal) cela revient à chercher tous les alignements possibles pour en extraire l'alignement optimal.

3.1.3 Réponse 6 : Complexité spatiale

La complexité spatiale de cet algorithme naïf est la même que sa complexité temporelle car d'après les questions précédentes , le nombre d'alignements possibles de (x, y) est en $O(2^n)$. Donc on aura besoin au minimum de 2^n cases pour stocker ces alignements puisque l'algorithme énumère tous les alignements de deux mots en vue de trouver la distance d'édition entre ces deux mots ou de trouver un alignement de coût minimal.

3.1.4 Tâche A :

1. Performances de la méthode :
   En moins d'une minute, on ne peut exécuter que les instances de taille inférieur ou égal à 12.



2. Consommation mémoire :

3.2 Programmation dynamique :

3.2.1 Calcul de la distance d'édition par programmation dynamique

Réponse 7 :

Soit $\sum = \{A, T, C, G\}$ un alphabet et $(\overline{u}, \overline{v})$ un alignement de $(x_{[1..i]}, y_{[1..j]})$ de longueur l.

1. Si $\overline{u_l} = -$ *alors* $\overline{v_l} \neq -$*donc* $\overline{v_l} \in \sum$, $\overline{v_l} \in y_{[1..j]}$.
2. Si $\overline{v_l} = -$ *alors* $\overline{u_l} \neq -$*donc* $\overline{u_l} \in \sum$, $\overline{u_l} \in x_{[1..i]}$.
3. Si $\overline{u_l} \neq -$ et $\overline{v_l} \neq -$ alors $\overline{v_l} \in \sum$ *et* $\overline{u_l} \in \sum$ou encore $\overline{u_l} \in x_{[1..i]}$ et $\overline{v_l} \in y_{[1..j]}$.

Justification :D'après la définition d'un alignement de deux mots :
$\forall i \in [1..l]$(l la longueur de l'alignement) $\overline{v_i} \neq -$ ou $\overline{u_i} \neq -$.
($\overline{u}$ et $\overline{v}$ ne présentent pas de gap à la même position).

Réponse 8 :
C($\overline{u}, \overline{v}$) $= C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}) + C(\overline{u}_l, \overline{v}_l)$.

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

Réponse 9 :

(x,y) un couple de mot de longueurs respectives n et m .
Pour $i \in$[1..n] *et* j $\in$ [1,m], $(\overline{u}, \overline{v})$ un alignement de $(x_{[1..i]}, y_{[1..j]})$ de longueur l.
On a : D(i,j)=d($x_{[1..i]}, y_{[1..j]}$).
D'après la définition de la distance d'édition de x à y : d(x,y) =min(C($\overline{x}, \overline{y}$)|($\overline{x}, \overline{y}$) est un alignement de (x,y)).
$Alors : D(i,j) = d(x_{[1..i]}, y_{[1..j]}) = min(C(\overline{u}, \overline{v}))$.
Des questions 7 et 8:on déduit que:
$D(i,j) = min\left\{C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}) + C(\overline{u}_l, \overline{v}_l)\right\}$

Et D(i,j)= min$\left\{(C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}))\right\} + C(\overline{u}_l, \overline{v}_l))$

Si C$(\overline{u}_l, \overline{v}_l) = c_{sub}(\overline{u}_l, \overline{v}_l)$:
$Comme\ \overline{u}_l \neq -$ et $\overline{v}_l \neq -$
$Alors$: le nombre de caractères sans tenir compte des gaps dans $\overline{u}_{[1..l-1]}$ est égale à $i-1$ puisqu'il y a i caractères dans $\overline{u}_{[1..l]}$.
Et $j-1$ caractères différents des gaps dans $\overline{v}_{[1..l-1]}$ puisqu'il y a j caractères dans $\overline{v}_{[1..l]}$.

On en déduit :
min$\left\{(C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}))\right\} = D(i-1, j-1)$, tel que $(i-1)$ la longueur de la séquence $x_{[1..i-1]}$ *et* $(j-1)$ la longueur de la séquence $y_{[1..j-1]}$.

Si C$(\overline{u}_l, \overline{v}_l) = c_{ins}$:
Cela implique :$\overline{u}_l = -\ et\ \overline{v}_l \neq -$.
De plus : le nombre de caractères différents du gap dans $\overline{u}_{[1..l-1]}$ est le même dans la séquence $\overline{u}_{[1..l]}$ qui est égale à i.
Le nombre de caractères différents du gap dans $\overline{v}_{[1..l-1]}$ est égale au nombre de caractères dans $\overline{v}_{[1..l]}$ moins un puisque $\overline{v}_l \neq -$ c'est à dire (j-1).
$Alors$:
($\overline{u}_{[1..l-1]}$, $\overline{u}_{[1..l-1]}$) est un alignement de $(\overline{x}_{[1..i]}, \overline{y}_{[1..j-1]})$.
Et min$(C(\overline{u}_{[1..l-1]}, \overline{u}_{[1..l-1]}))$= d$(\overline{x}_{[1..i]}, \overline{y}_{[1..j-1]}) = D(i, j-1)$.
$SiC(\overline{u}_l, \overline{v}_l) = c_{del}$:
Cela implique :$\overline{v}_l = -\ et\ \overline{u}_l \neq -$.
De plus :
Le nombre de caractères différents du gap dans $\overline{v}_{[1..l-1]}$ est le même dans la séquence $\overline{v}_{[1..l]}$ qui est égale à j.
Le nombre de caractères différents du gap dans $\overline{u}_{[1..l-1]}$ est égale au nombre de caractères dans $\overline{u}_{[1..l]}$ moins un puisque $\overline{u}_l \neq -$ c'est à dire (i-1).
$Alors$:
($\overline{u}_{[1..l-1]}$, $\overline{u}_{[1..l-1]}$) est un alignement de $(\overline{x}_{[1..i-1]}, \overline{y}_{[1..j]})$.

Et $\min(C(\overline{u}_{[1..l-1]}, \overline{u}_{[1..l-1]}))$= d$(\overline{x}_{[1..i-1]}, \overline{y}_{[1..j]}) = D(i-1, j)$.

De la on peut déduire que :

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

Réponse 10 :
$D(0,0) = 0$.
Justification :
$D(0,0)$ est la distance d'édition de deux séquences de longueur nulle.Comme la longueur est nulle,le coût d'alignement de ces deux séquences est nul , cela implique que la distance d'édition est nulle $D(0,0) = 0$.

Réponse 11 :
D(0,j)=2j (2 est le coût d'insertion).
$D(0,j)$: correspond à la distance d'édition de deux mots x et y de longueurs respectives $0, j$.
Pour obtenir un alignement optimal, on a qu'un seul choix,insérer successivement dans x des caractères de y jusqu'à l'obtention de la séquence y.
Au total on aura j insertions car la séquence y possède j caractères .

$D(i,0) = 2i$ (2 le coût d'insertion).
D(i,0) :correspond à la distance d'édition de deux mots x et y de longueurs respectives $i, 0$.
Pour obtenir un alignement optimal, on a qu'un seul choix,effectuer des suppressions successives dans x jusqu'à l'obtention de la séquence y.
Au total on aura i suppressions car la séquence i possède i caractères et la séquence y 0 caractères.

Réponse 12 :Un algorithme itératif Algorithme DIST-1 :
1) Calcul du coût du chemin optimal(**distance d'édition** qui aboutit sur chaque nœud (i, j) en partant du nœud $(0, 0)$,jusqu'au nœud $[N, M]$ de notre grille.
2) Enregistrer ces coûts dans un tableau de dimension $N \times M$.

Comment va-t-on calculer ces coûts ?

Pseudo-code de DIST_1

Algorithm 1 DIST_1

entrée: x et y deux mots de longueurs respectives n et m
sortie: La distance d'édition de x à y
debut
  T *un tableau à 2 dimensions : n lignes* $\times m$ *colonnes*
  T[0][0] $\leftarrow 0$
  pour i *de* 0 *à* $n+1$ **faire**
    pour j *de* 0 *à* $m+1$ **faire**
      si $i \neq 0$ *et* $i \neq 0$ **alors**
        $T[i][j] \leftarrow minimum(c_{ins} + T[i][j-1], c_{del} + T[i-1][j], c_{sub}(x[i-1], y[j-1]) + T[i-1][j-1])$;
      sinon
        si $i=0$ **alors**
          $T[0][j] \leftarrow 2 * j$
        si $j=0$ **alors**
          $T[i][0] \leftarrow 2 * i$
  retourner $T[n][m]$
  fin

Réponse 13 :Complexité spatiale

Le calcul de la distance d'édition par la fonction DIST-1 nécessite la création d'une matrice à deux dimensions donc une réservation d'un espace mémoire de l'ordre de $N \times M$ cases mémoires. La complexité spatiale de cette fonction est équivalente O(N^2),*unecomplexitéquadratique*.

Réponse 14 :Complexité temporelle

Pour mesurer les performances en temps de l'algorithme,on calcule le nombre d'opérations(Comparaisons et calculs) qui doivent être effectuées pour que l'algorithme aboutisse à son résultat.

Pour deux séquences de longueurs respectives N et M, le nombre de comparaisons et de calculs effectués par l'algorithme DIST_1 est de l'ordre de $O(N * M)$.On dit que la complexité est quadratique car on parcourt la matrice de taille $N * M$($(N * M)$ tours de boucle) et pour chaque tour de boucle on effectue 3 comparaisons de complexité constante en $\theta(1)$donc on a une complexité de l'ordre de $3 \times N \times M$,elle est équivalente à N^2 (une complexité quadratique).

3.2.2 Calcul d'un alignement optimal par programmation dynamique

Réponse 15 :
Soit (i,j) $\in [0, n] \times [o, m]$.
$Al^*(i,j) = \{(\overline{u}, \overline{v})$, $(\overline{u}, \overline{v})$ un alignement de $(x_{[1..i]}, y_{[1..j]})telqueC(\overline{u}, \overline{v}) = d(x_{[1..i]}, y_{[1..j]})$.
$Montronsque$:

$SiD(i,j) = D(i-1,j-1) + c_{sub}(x_i, y_j),\ alors (\overline{s}, \overline{t}) \in Al^*(i-1,j-1)$,
$(\overline{s}.x_i, \overline{t}.y_j) \in Al^*(i,j)$.

Par l'absurde :
On suppose que $D(i,j) = D(i-1,j-1) + c_{sub}(x_i, y_j)$,
alors $\exists(\overline{r}, \overline{q}) \notin$Al$^*(i-1,j-1)$ tel que $(\overline{r}.x_i, \overline{q}.y_j) \in Al^*(i,j)$.
$SoitD(i,j)ladistanced'éditiondex_{[1..i]}$ à $y_{[1..j]}$ et $(\overline{u}, \overline{v})$ l'alignement optimal de longueur l associé à $D(i,j)$.
Montrons par absurde que $(\overline{s}, \overline{t})$ est un alignement optimal de longueur $(l-1)$ associé à D(i-1,j-1).
On suppose qu'il existe un alignement $(\overline{r}, \overline{q})$ de longueur $(l-1)$ qui n'est pas optimal et que $(\overline{r}.x_i, \overline{q}.y_j)$ est un alignement optimal associé à D(i,j).
On a : $(\overline{u}, \overline{v}) = (\overline{r}.x_i, \overline{q}.y_j)$.
Et : C$((\overline{u}, \overline{v})$=C$((\overline{r}, \overline{q})) + c_{sub}$.
$Etcomme(\overline{u}, \overline{v}) \in Al^*(i,j)$:
$Alors : C(\overline{u}, \overline{v})$= D(i,j).
Par substitution, on obtient : D(i,j)=C$((\overline{u}, \overline{v})) = C((\overline{r}, \overline{q})) + c_{sub}.(Contradiction)$
$Dufaitque(\overline{r}, \overline{q}) \notin$Al*(i-1,j-1) alors $C(\overline{r}, \overline{q}) > d(x_{[1..i-1]}, y_{[1..j-1]})et$
$C(\overline{r}, \overline{q}) > D(i-1,j-1)$.
On en déduit que $(\overline{r}, \overline{q})$ est forcément un alignement optimal de longueur $(l-1)$ associé à $D(i-1,j-1)$ vérifiant $(\overline{r}, \overline{q}) \in$Al$^*(i-1,j-1)$ et
$(\overline{r}.x_i, \overline{q}.y_j) \in$Al$^*(i,j)$.
On suivant le même raisonnement , on pourra également montrer que :
Si D(i,j)=D(i,j-1)+ c_{ins}, $alors(\overline{s}, \overline{t}) \in$Al$^*(i,j-1)$, $(\overline{s}.-, \overline{t}.y_j) \in$Al$^*(i,j)$.
$SiD(i,j) = D(i-1,j) + c_{del},\ alors(\overline{s}, \overline{t}) \in Al^*(i-1,j), (\overline{s}.x_i, \overline{t}.-) \in$Al$^*(i,j)$.

Réponse 16 : $Pseudo\_code$ **de SOL**_1

Algorithm 2 SOL_1

entrée: x et y deux mots de longueurs respectives n et m
T : un tableau à deux dimensions renvoyé par la fonction $Dist\_1$ contenant toutes les valeurs de D
sortie: un alignement optimal(de coût minimal) $(\overline{x}, \overline{y})$ de (x, y)
début

```
alignX ← ""
alignY ← ""
i ← n
j ← m
tant que (i ≠ 0 ou j ≠ 0) faire
    (indexX, indexY) ← minimum(T, i, j, x, y) rend l'indice de la case de coût minimal de
     ces trois cases suivantes : T[i − 1][j − 1] et T[i][j − 1] et T[i − 1][j]
     si (indexX= i − 1 and indexY= j − 1) alors
         alignX ← x[i − 1] + alignX
         alignY ← y[j − 1] + alignY
    sinon
         si (indexX=i and indexY= j − 1) alors
             alignX ← − + alignX
              alignY ← y[j − 1] + alignY
         si (indexX = i − 1 and indexY = j) alors
             alignX ← x[i − 1] + alignX
              alignY ← − + alignY
     i ← indexX
     j ← indexY
retourner (alignX, alignY)
```

fin

Réponse 17 :Complexité temporelle du problème d'alignement :
Trouver un alignement optimal de deux mots x et y de longueur N et M respectives est basé sur deux phases :

L'algorithme $DIST\_1$:Calcule toutes les valeurs D(i,j),les stockées dans un tableau à deux dimensions avec une complexité temporelle en O(N×M)(**Question 14)**.

L'algorithme SOL_1 : parcourt le tableau D renvoyé par $DIST\_1$ de la dernière case $D(N, M)$ vers la première case $D(0, 0)$ en prenant à chaque fois la case de coût le plus faible parmi les trois cases voisines($D(i, j-1)$, $D(i-1, j-1)$, $D(i, j-1)$) donc à chaque tour de boucle, on effectue trois lectures de case en $\theta(1)$ et trois opérations élémentaires(comparaisons) de coût constant et on construit simultanément l'alignement de la droite vers la gauche avec deux affectations de coût constant.

Dans le pire des cas, on parcourt (N+M) cases du tableau D (la longueur maximale d'un alignement de (x,y)) on effectuant à chaque tour de boucle trois comparaisons et deux affectations de coût constant. On en déduit que la complexité temporelle de la fonction SOL_1est de l'ordre de $O(5 \times (N + M))$ qui est équivalent à O(N+M).

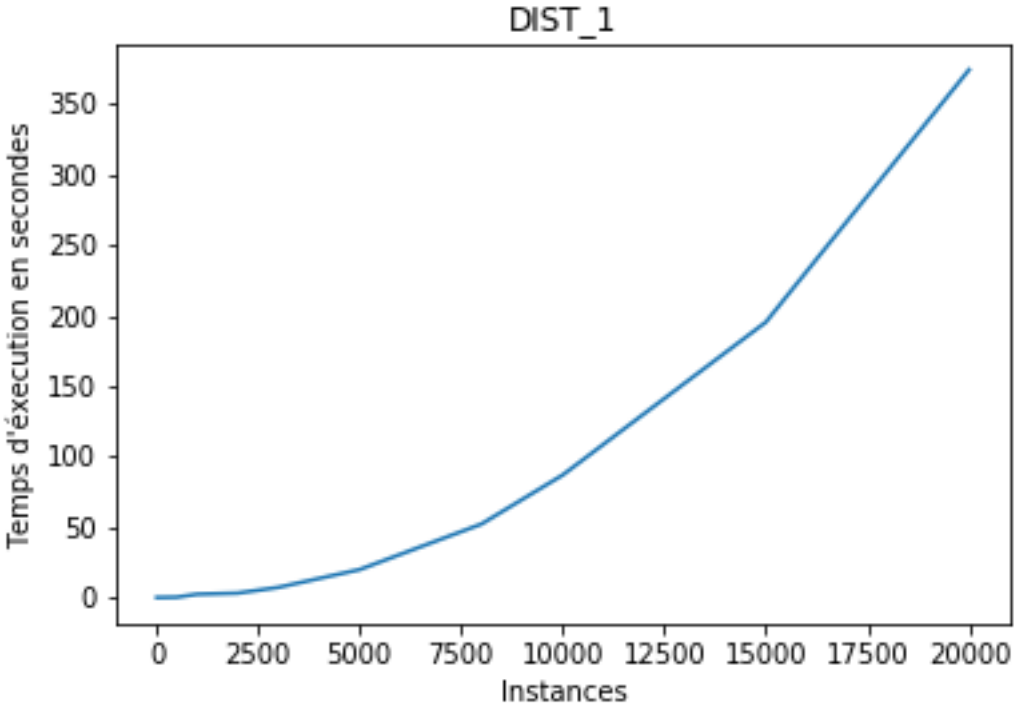
On résout le problème d'alignement en combinant les deux algorithmes $DIST\_1$ et SOL_1 avec une complexité temporelle de l'ordre de O($(N \times M) + (N + M)$) équivalente à $O(N \times M)$ ou même $O(N^2)$ une complexité quadratique.

Réponse 18 :Complexité spatiale du problème d'alignement :

Comme ce problème se résout en combinant l'algorithme $DIST\_1$ de complexité spatiale $\theta(N \times M)$, ($(N \times M)$ cases mémoires réservées pour la matrice qui stockera au fur et à mesure toutes les valeurs possibles de D) avec l'algorithme SOL_1 de complexité spatiale $O(2\times(N + M))$ qui correspond aux deux tableaux de taille maximale (N+M) qui stockeront l'alignement obtenu des deux mots passés en paramètre, dans le pire des cas(l'alignement a une longueur maximale de taille $N \times M$).
On déduit que la complexité spatiale de ce problème d'alignement est en O($(N \times M)$ +(N+M)) équivalent à O($(N \times M)$).

3.2.3 Tâche B :

— Courbe de la consommation de temps CPU de l'algorithme DIST$en fonction de la taille du premier m$



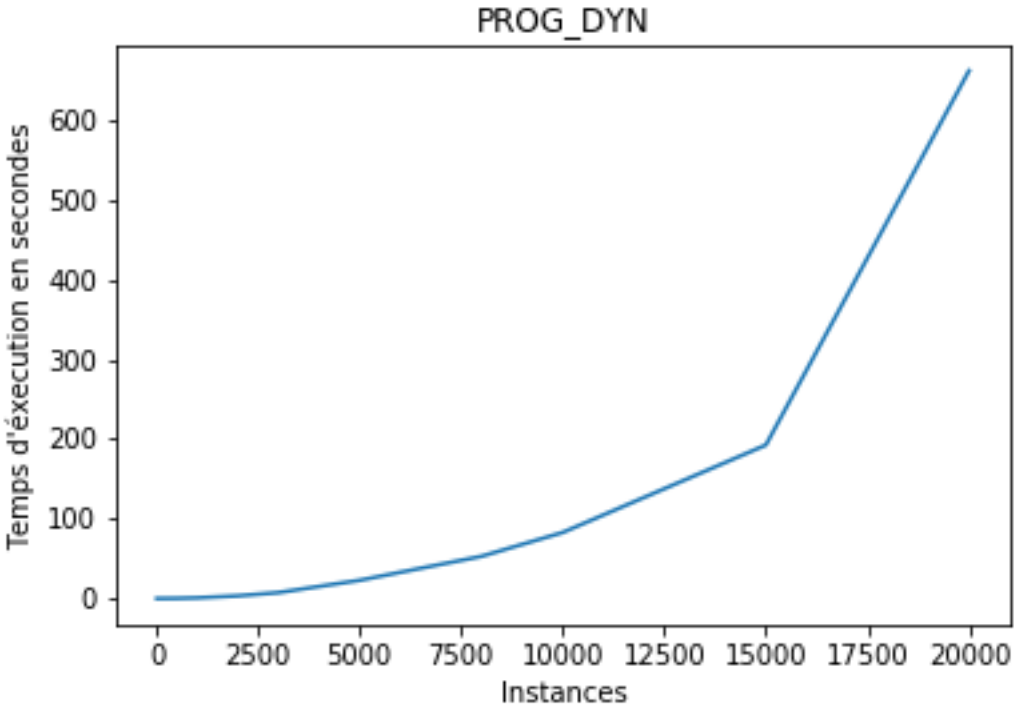
Montrons que la courbe obtenue correspond à la complexité théorique :
Soit un tableau temps-exécution qui stockera le coût d'exécution de chaque instance fournie, tel que n_i la taille de la première séquence et t_i le temps d'exécution de cette instance.

$s10$	$s12$	$s13$
$t10$ 0.0000000058	$t12$ 0.0001490	$t13$ 0.00023436
$s14$	$s15$	$s20$
$t14$ 0.000217437744140625	$t15$ 0.0002837181091308594	$t20$ 0.0003981590270996094
$s50$	$s100$	$s500$
$t50$ 0.0020177364349365234	$t100$ 0.00884921646118164	$t500$ 0.1917037

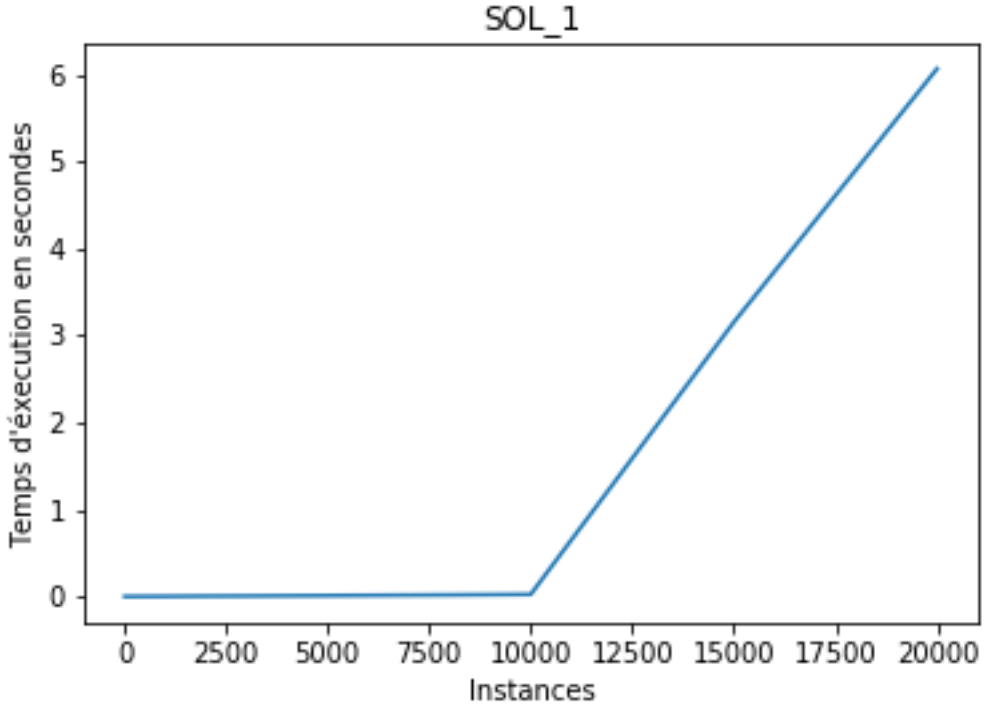
Si on prend n=20 et $n1$=100= 5×n
On constate que : 0.0003981590270996094 ×25 ≈ $t100$.
0.0003981590270996094 × 5^2 ≈ $t100$.
De même si on prend n=50 et $n1$=100= 2×n.
On constate que : 0.0020177364349365234 ×4=0.00919553976 ≈t100.
0.0020177364349365234 ×2^2 = 0.008070946156732 ≈t100.
On choisissant plusieurs valeurs de n,la propriété précédente est vérifiée , on en déduit qu'il s'agit d'un parcours de tableaux à deux dimensions .
La courbe obtenue correspond bien à la complexité théorique de la fonction $DIST\_1$ puisqu'il s'agit d'un parcours de tableaux à deux dimensions .

— Estimation de la quantité de mémoire utilisée par PROG_DYN pour une instance de très grande taille :
Pour une séquence de taille égale à 50000, Le noyau est mort et le système stoppe le programme.
En utilisant la méthode top sur linux, la quantité de mémoire utilisée par PROG_DYN pour une instance de très grande taille supérieur à 50000 est de l'ordre de 7070964.1 KIB

— Courbe de la consommation de temps CPU de la l'algorithme $PROG\_DYN$ en fonction de la taille du premier mot du couple des instances fournies .



— Courbe de la consommation de temps CPU de la l'algorithme SOL_1 en fonction de la taille du premier mot du couple des instances fournies .



3.2.4 Amélioration de la complexité spatiale du calcul de la distance

Réponse 19 :
Lors du remplissage de la ligne i > 0 du tableau T dans l'algorithme DIST-1, il suffit d'avoir accès aux lignes i-1 et i du tableau(partiellement remplie pour cette dernière) car :

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

Le tableau T est rempli de haut en bas et de gauche à droite.Pour tout i > 0 , pour calculer D(i,j) d'après la question 9 et 15, on aura besoin des cases $(i-1, j-1)$, $(i, j-1)$ et de la case (i-1,j) de T , ces cases sont situées soit à gauche de la case[i,j] soit en dessus (la ligne d'avant donc $(i-1)$).

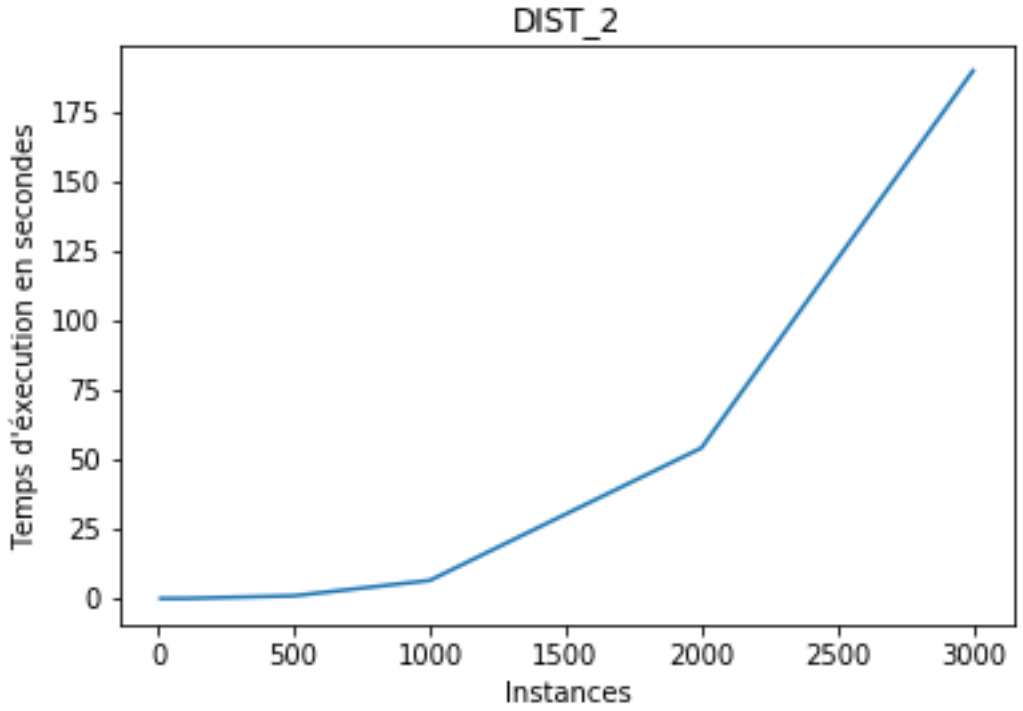
Réponse 20 :Pseudo-code de DIST_2

Algorithm 3 Dist_2

entrée: x et y deux mots de longueurs respectives n et m
sortie: la distance d'édition de x à y
début
    T *un tableau de taille* m
    tmp1, tmp2 ← *deux variables temporelles*
    pour j *de* 0 *à* $m+1$ **faire**
        T[j] ←$2\times j$
    pour i *de* 1 *à* $n+1$ **faire**
        pour j *de* 0 *à* $m+1$ **faire**
            $tmp1 \leftarrow T[j]$
            si $(j \neq 0)$ **alors**
                $min \leftarrow minimum(c_{ins} + T[j-1], c_{del} + T[j],$
                $c_{sub}(x[i-1], y[j-1]) + tmp2)$
                $T \leftarrow T[0:j] + [min] + T[(j+1):]$
            sinon
                $T \leftarrow T[0:i] + T[1:]$
            $tmp2 = tmp1$
    retourner $T[n][m]$
fin

3.2.5 Tâche C

— La courbe de consommation de temps CPU en fonction de la taille du premier mot du couple des instances fournies.



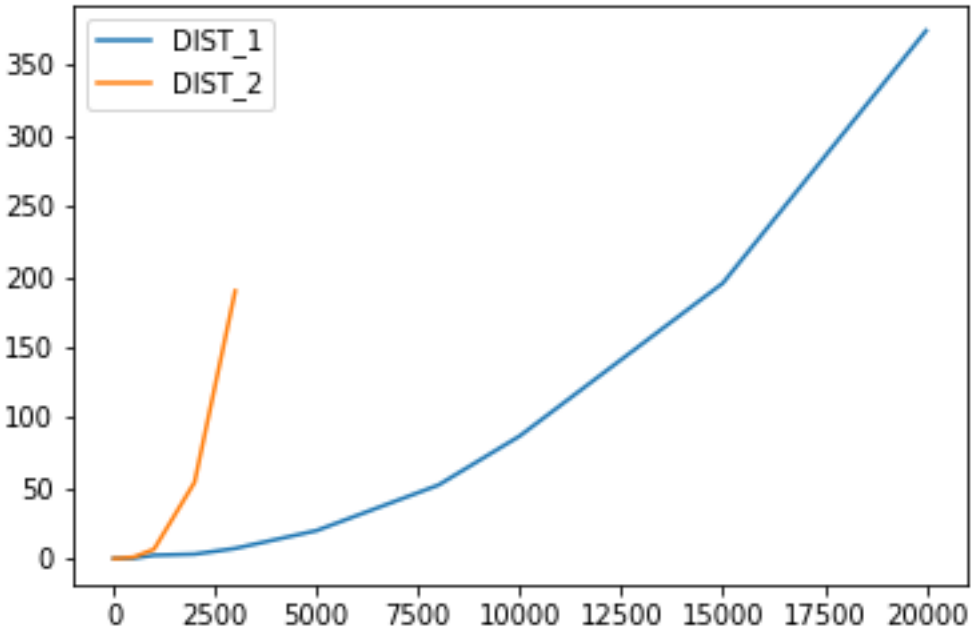
Montrons que la courbe obtenue correspond à la complexité théorique :
Soit un tableau temps-exécution qui stockera le coût d'exécution de chaque instance fournie, tel que n_i la taille de la première séquence et t_i le temps d'exécution de cette instance.

n10	n12	n13
t10 0.000125885009765625	t12 0.0001323223114013672	t13 0.000186920166015625
n14	n15	n20
t14 0.000156402587890625	t15 0.00035953521728515625	t20 0.0003948211669921875
n50	n100	n500
t50 0.0027022361755371094	t100 0.01643991470336914	t500 0.9551939964294434

Si on prend n=50 et $n1$=1000 alors $t_{n1} \neq 2 \times t_n$
De même que si : n =100 etn1=500alors$t_{n1} \neq 5 \times t_n$
On testant plusieurs valeurs de n comme précédemment ,et en observant l'allure du graphe, on déduit que l'algorithme est de complexité spatiale linéaire(utilisation d'un

tableau à une dimension) mais de complexité temporelle polynomiale en raison des traitements effectués au sein de la boucle qui parcourt le tableau et modifie ces valeurs de sorte qu'il ne perd pas d'informations nécessaires pour le calcul de D(j), en stockant dans des variables temporelles les valeurs de D à protéger.

— Comparaison entre DIST_1 et $DIST\_2$



Du point de vue de la complexité spatiale, l'algorithme DIST_2est plus optimal que l'algorithme DIST_1car il utilise un seul tableau pour stocker les valeurs de D.

Par contre en temps de calcul, l'algorithme DIST_1 est plus rapide et efficace que DIST_2, il permet de calculer la distance d'édition de deux mots de longueurs inférieur ou égale à 20000 en réservant 20000^2 cases mémoires , tant dis que DIST_2 nous permet de calculer que la distance d'édition des mots de longueur inférieur ou égale à 5000 en réservant seulement 5000 cases mémoires .
En utilisant DIST_2, on gagne en mémoire mais on perd beaucoup en temps de calcul à cause des traitements effectués au sein de la boucle qui parcourt le tableau et modifie ces valeurs de sorte qu'il ne perd pas d'informations nécessaires pour le calcul de D(j) .

— En utilisant la méthode top sur linux, la quantité de mémoire utilisée par DIST_2 pour une instance de très grande taille est de l'ordre de 14893,28 KIB.

3.3 Amélioration de la complexité spatiale du calcul de la distance :

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

3.4.1 Réponse 21 :

Le pseudo-code de la méthode mot_gaps :

Algorithm 4 mot-gaps

entrée: k entier naturel
sortie: chaîne de caractères constituée de k gaps
debut
    $s \leftarrow$ ' ' *:chaîne vide*
    pour $i = 1$ *à* k **faire**
        $s \leftarrow s+$'$-$'
    retourner s
fin

3.4.2 Réponse 22 :

Algorithm 5 align_lettre_mot(x, y)

entrée: x un mot de longueur 1, y un mot de longueur m quelconque
sortie: un alignement minimal $(\overline{x}, \overline{y})$ de (x, y)
debut
    $c =$' ' *une chaîne de caractères initialisée à vide*
    i *un compteur initialisé à* 0
    tant que $i < m$ *et* $x[0] \neq y[i]$ **faire**
        $c \leftarrow c$+'−'
        $i \leftarrow i + 1$
        On sort de la boucle si on a trouvé un élément en commun entre x *et* y *ou si on a finit de parcourir tout le tableau*
    Dans le premier cas, on fait des insertions avec les éléments restants du tableau
    $s \leftarrow$ ' '
    si $i < m$ *et* $x = y[i]$ **alors**
        pour $k = i + 1$ *jusqu'à* $m - 1$ **faire**
            $s \leftarrow s$+'−'
            $k \leftarrow k + 1$
        $c \leftarrow c + x + s$
    sinon on aura fait des insertions tout le long, l'élément de x *n'est pas dans* y*, on doit faire une substitution*
    sinon
        $c \leftarrow c_{[0..m-1]} + x$
    retourner (c, y)
fin

3.4.3 Réponse 23 :

Les alignements optimaux possibles pour $(\overline{s}, \overline{t})$ sont :
$\overline{s} = BAL$ et $\overline{t} = RO-$ (2 substitutions puis une supression
ou
$\overline{s} = BAL$ et $\overline{t} = R - O$ (1 substitution puis 1 supression puis 1 substitution)
ou
$\overline{s} = BAL$ ou $\overline{t} = -RO$ (1 supression puis 2 substitutions)
La distance est donc : **13**
Les alignements possible pour $(\overline{u}, \overline{v})$ sont :
(2 supressions et 1 insertion))
$\overline{u} = LON-$ et $\overline{v} = --ND$
La distance est donc : 9

Pour montrer que $(\overline{s}.\overline{u}, \overline{t}.\overline{v})$ n'est pas un alignement optimal, il me suffit de trouver un alignement de distance inférieure :
Avec le coupage en deux :

$dist(x, y) = dist(x^1, y^1) + dist(x^2, y^2) = 13 + 9 = 22$
Si on pose :
$\overline{x} = BALLON-$ et $\overline{y} = R---OND$
(1 substitution puis 3 supressions puis 1 insertion)
$dist(x, y) = 5 + 3 * 3 + 3 = 17 < 22$
Conclusion : $(\overline{s}.\overline{u}, \overline{t}.\overline{v})$ n'est pas un alignement optimal.

3.4.4 Réponse 24 :

Algorithm 6 SOL_2 : Algorithme diviser pour régner pour le calcul d'un alignement optimal

entrée: x chaîne de caractères de longueur n, y chaîne de caractères de longueur m
sortie: un alignement minimal $(\overline{x}, \overline{y})$ de (x, y)
debut

```
    i ← n/2                                                          fin
    cas de base
    Soit y est vide, on ne doit faire que des suppressions dans x, donc on retourne la chaine x
    avec des gaps dans y
    si m = 0 alors
    | retourner (x, mot_gaps(n))
   Soit x a une seule lettre, on se retrouve décrit dans la fonction align_lettre_mot
    si n = 1 alors
    |  retourner align_lettre_mot(x, y)
    |  sinon, on doit trouver la bonne coupure de y et faire des appels récursifs
    |  sinon
    |  |  j = coupure(x, y)
    |  |  A = SOL_2(x_[0..i], y_[0..j])
    |  |  B = SOL_2(x_[i..n], y_[j..m])
    |  |  retourner (A[0] + B[0], A[1] + B[1])
```

3.4.5 Réponse 25 :

Algorithm 7 coupure

entrée: x un mot de longueur n, y un mot de longueur m
sortie: coupure j associée à $i^* = len(x)/2$
debut

$I1 \leftarrow [0, 1, ..., m]$ *Initialisation de I1 à l'indice de la colonne où le chemin optimal de* (i^*, j) *à* $(0, 0)$ *croise la ligne* i^*
$I2 \leftarrow [0, 1, ..., m]$ *Initialisation de I2*

initialisation de $D1$ *avec les valeurs associées du tableau de l'algorithme dynamique de la partie précédente*
$D1 \leftarrow$[1,2,4,...,2m]
$D2 \leftarrow$[0]*m

pour $i = 1$ *à* n **faire**
  $D2[0] \leftarrow 2 * i$
  initialisation de la première colonne
  pour $j = 1$ *jusqu'à* $j = m$ **faire**
    $D2[j] \leftarrow$le minimum de $(c_{ins}+D2[j-1], c_{del}+D1[j], c_{sub}(x[i-1], y[j-1])+D1[j-1])$
  Selon l'algorithme de programmation dynamique de la partie précédente
  On remplit le tableau I au fur et à mesure selon le cout optimal de l'alignement
  si $i > x/2$ **alors**
    pour $j = 0$ *à* m **faire**
      si $(D2[j] = c_{ins} + D2[j-1]$ **alors**
        $I2[j] \leftarrow I2[j-1]$
      sinon
        si $D2[j] = c_{del} + D1[j]$ **alors**
          $I2[j] \leftarrow I2[j]$
        sinon
          si $D2[j] = c_{sub(x[i-1],y[j-1])} + D1[j-1]$ **alors**
            $I2[j] \leftarrow I1[j-1]$
  On range les valeurs de la deuxième liste de D dans la première, De même pour I
  pour $j = 0$ *à* m **faire**
    $D1[j] \leftarrow D2[j]$
  si $i > n/2$ **alors**
    pour $j = 0$ *à* m **faire**
      $I1[j] \leftarrow I2[j]$
retourner $I2[m]$

fin

3.4.6 Réponse 26 :

compléxité spatiale de coupure :

La fonction coupure utilise 2 tableau D et I de taille $|y|+1$ chacun, 2 variables lx et ly pour stocker les valeurs $|x|$ et $|y|$, et deux variables i et j pour le parcours des tableaux, On peut compter la mémoire occupée par les copies de x et y qui est en $|y|+|x|$. La complexité spatiale est donc d'environ $(4+2(|y|+1)+|x|+|y|) = (6+3|y|+|x|)$.
r Si on considère $|y| = m$ et $|x| = n$ alors, la complexité spatiale de coupure est de l'ordre de $O(3m+n) = O(m+n)$

3.4.7 Réponse 27 :

complexité spatiale de SOL_2 :
Dans la fonction SOL_2, on a 4 variables entières et 2 variables qui stockent les retours des appels récursifs de SOL_2.
La complexité spatiale de l'appel de mot_gaps est en $O(n)$
La complexité spatiale de l'appel de align_lettre_mot est en $O(n+2m)$ (O(n+m) pour la copie de x et y, $O(m)$ pour les listes m et s. Elle est donc de l'ordre de **O(m+n)**
La complexité spatiale de coupure étant de $O(3m+n)$, on se retrouve avec un total d'environ $O(5m+2n)$ par appel récursif.
$h = log_2 n$ la hauteur de l'arbre de récursion
Le nombre d'appels récursifs est donné par $\sum_{i=0}^{log_2 n} 2^i = 2^{log_2 n+1} - 1 = 2n-1$
n_i , m_i étant la taille des chaines x_i ,y_i données en paramètre au ième appel récursif
La complexité spatiale totale peut être calculée comme suit :
$\sum_{i=1}^{2n-1}(5m_i + 2n_i)$ et on sait que la taille de x est divisé sur 2 dans chaque appel récursif à l'intérieur d'une fonction. On peut dire que dans l'arbre de récursion $n_i = \frac{n}{2^i}$ au i ème étage de l'arbre avec $i \in [0,h]$. A chaque étage on aurait 2^i appels récursifs sur des tableaux x_i de même taille.
J'en viens à la formule finale suivante quand à la complexité spatiale :
$C = \sum_{i=1,k=0}^{i=2n-1,k=log_2 n}(5m_i + 2\frac{n}{2^k})$
avec (k=k+1) ssi i est une puissance de 2
$\forall i \in [0, 2n-1]$ $m_i <= m$ et $n_i <= n$ donc $C < (2n-1) * (5m+2n)$ donc $C < O(mn)$, On a donc amélioré la complexité spatiale si on compare SOL_2 à SOL_1.

3.4.8 Réponse 28 :

complexité temporelle de coupure :
$|x| = n, |y| = m$ dans ce qui suit
Initialisation des tableaux I et D en $O(4 * (m+1))$ La première boucle a (n+1) itération et comporte 2 boucles internes à (m+1) itérations (la première et la troisième) et 2 boucles interne à (m+1) itérations exécutée par la boucle externe (n/2) fois seulement.

On a $4(m+1)+(n+1)*(2(m+1))+(\frac{n+1}{2})*2(m+1) = 4m+4+3(n+1)(m+1) = 7m+7+3nm+3n$
La complexité temporelle de coupure est de l'ordre de **O(mn)**

3.4.9 Réponse 29 :

En terme de complexité, on se retrouve aussi avec $O(mn)$ car la complexité de coupure est dominante sauf qu'en terme de temps, SOL_2 serait plus longue car coupure est appelé un grand nombre de fois à travers les appels récursifs. On peut constater ça expérimentalement dans le graphe ci-dessus (je n'arrive pas à l'afficher après la réponse 29 même si j'ai bien placé la figure après la question 29 sur le code latex).
Les résultats experimentales sont en accord avec la complexité théorique.
Remarques :
Estimation de la consommation mémoire :
J'ai testé $\text{SOL}_2 pour une instance ayant 20000 caractères afin de l'utiliser pour calculer top, au bout de 35 min, ell$
$Pour une instance de 8000 caractères on a une consommation mémoire de 27442 KiB$
$Pour une instance de 10000 caractères on a une consommation mémoire de 62725 KiB$

4 Une extension : l'alignement local des séquences

4.0.1 Réponse 30 :

J'utilise une fonction pour générer des mots aléatoirement, voici les résultats de 6 instances générées avec celle ci. Le calcul du coût est fait avec la fonction DIST_1

x	y	coût minimal	$(\|x\| - \|y\|)c_{del}$
GATGACTTCTACAATTATGA	AGT	34	(20-3)*2=34
ATCATAGCAAAGAGCGTTACGGGCAAATAC	AGCC	52	(30-4)*2=52
TGTTGGGCAGGTATGGACGCCAGGAGAGGT	ATACA	50	(30-5)*2=50
AGATAAGGCGGAATTGGGTCTCATA	AAATC	40	(25-5)*2=40
GTACTTGTCCAATCTTCCGAATT	TGTATA	34	(23-6)*2=34
GGACTTAGCACGATAAAGAAAGA	TATAAA	34	(23-6)*2=34

On constate bien que pour des mots y de tailles petites relativement à celle de y, on a :
coût d'un alignement global (x,y) $= |x| - |y|c_{del}$

4.0.2 Réponse 31 :

Ce n'est pas une bonne idée, car si on prend l'exemple de coût maximal donné dans la question 2 :
Exemple : Si on prend $x =$ "AGT", $y =$ "$TACG$"

$|x| = 3$ et $|y| = 4$
Soit un alignement qui suit : $\overline{x} =$ "$AGT - - - -$" et $\overline{y} =$ " $- - - TACG$"
Si considère que le coût des insertions et suppressions en début et fin de mots valent 0, le coût de cet alignemenet vaudrait 0. Ce qui nous amène à une contradiction car c'est un alignement de coût maximal.

4.0.3 Réponse 32 :

Complexité spatiale :
En s'inspirant de la partie 3, dans l'algorithme dynamique, il suffit de modifier les coût des insertions, suppressions et substitutions comme indiqué pour calculer le score.
Comme indiqué dans la formule de BEST_SCORE, on applique cet algorithme de calcul du score : SCORE pour tous les facteurs possibles de même taille x et de y nb avec.
Soit $s = min(m, n)$ $nb = \sum_{k=0}^{s}(n - k + 1)(m - k + 1)$
Ce qui est de l'ordre de $O(smn)$ qui est équivalent à $O(mn^2)$
Donc Si on considère que la mémoire est restituée (libérée) après chaque appel de Score, alors la compléxité spatiale est bien en $O(n * m)$ sinon on multiplie cette compléxité par le nombre d'alignements possibles qui est de l'ordre de $O(mn^2)$ (voir le calcul ci-dessus) donc dans ce cas la complexité spatiale serait de $O(n^3m^2)$.
La complexité temporelle : Elle est donnée par le nombre d'appel de SCORE * La complexité de la fonction SCORE qui est de même complexité que DIST qui est en $O(nm)$, donc de même la complexité temporelle est $O(n^3m^2)$

Compléxité temporelle : On ne peut pas diminuer la complexité spatiale avec une méthode diviser pour régner sans dégrader la complexité spatiale car on se retrouvera dans le même problème que celui de la partie 3.