1. Le problème d'alignement de séquences

1.1 Alignement de deux mots

Q1.1.1 Montrer que si $(\bar{x}, \bar{y})$ et $(\bar{u}, \bar{v})$ sont respectivement des alignements de (x, y) et (u, v), alors $(\bar{x}.\bar{u}, \bar{y}.\bar{v})$ est un alignement de $(x.u,\ y.v)$.

Par définition :

On dit que $(\bar{x}, \bar{y})$ est un alignement de (x, y) ssi $\begin{cases} (i) & \pi(\bar{x}) = x \\ (ii) & \pi(\bar{y}) = y \\ (iii) & |\bar{x}| = |\bar{y}| \\ (iv) & \forall i \in [1..|\bar{x}|],\ \bar{x}_i \neq - \ ou\ \bar{y}_i \neq - \end{cases}$

Donc il faut montrer que :

$(i')\ \ \pi(\bar{x}.\bar{u}) = x.u$

$(ii')\ \ \pi(\bar{y}.\bar{v}) = y.v$

$(iii')\ \ |\bar{x}.\bar{u}| = |\bar{y}.\bar{v}|$

$(iv')\ \ \forall k \epsilon [1 \dots |\bar{x} \cdot \bar{u}|],\ (\bar{x} \cdot \bar{u})_i \neq - \ ou\ \ (\bar{y} \cdot \bar{v})_i \neq -$

On sait que $\pi(\bar{x}.\bar{u})$ correspond au mot au sous mot obtenu en supprimant tous les gaps de $\bar{x}.\bar{u}$, donc $\pi(\bar{x}.\bar{u}) = \pi(\bar{x})\pi(\bar{u}) = x.u$ d'après *(i)*

De même $\pi(\bar{y}.\bar{v})$ correspond au mot au sous mot obtenu en supprimant tous les gaps de $\bar{y}.\bar{v}$, donc $\pi(\bar{y}.\bar{v}) = \pi(\bar{y})\pi(\bar{v}) = y.v$ d'après *(ii)*

On a également $|\bar{x}.\bar{u}| = |\bar{x}| + |\bar{u}| = |\bar{y}| + |\bar{v}| = |\bar{y}.\bar{v}|$ d'après *(iii)*

Soit $k \epsilon [1 \dots |\bar{x} \cdot \bar{u}|]$ c'est-à-dire $k \epsilon [1 \dots m] \cup [m + 1 \dots n]$ avec m = $|\bar{x}|$ et n = $|\bar{u}|$, montrons que $(\bar{x} \cdot \bar{u})_i \neq - \ ou\ \ (\bar{y} \cdot \bar{v})_i \neq -$

On a :

Pour $k \epsilon [1 \dots m]$ on a : $\begin{Bmatrix} (\bar{x} \cdot \bar{u})_k = \bar{x}_k \\ (\bar{y} \cdot \bar{v})_k = \bar{y}_k \end{Bmatrix}$ or d'après *(iv)* on a : $\bar{x}_k \neq - \ ou\ \bar{y}_k \neq -$

Pour $k \epsilon [m + 1 \dots n]$ on a : $\begin{Bmatrix} (\bar{x} \cdot \bar{u})_k = \bar{u}_k \\ (\bar{y} \cdot \bar{v})_k = \bar{v}_k \end{Bmatrix}$ or d'après (iv) on a : $\bar{u}_k \neq - \ ou\ \bar{v}_k \neq -$

D'où le résultat attendu $\forall k \epsilon [1 \dots |\bar{x} \cdot \bar{u}|],\ (\bar{x} \cdot \bar{u})_k \neq - \ ou\ \ (\bar{y} \cdot \bar{v})_k \neq -$

Q.1.1.2 Si $x \epsilon \sum^*$ est un mot de longueur n et $y \epsilon \sum^*$ est de longueur m, quelle est la longueur maximale d'un alignement de (x, y)

La longueur maximale d'un alignement de deux mots x et y de taille respectivement n et m est de n+ m. on obtient cette longueur en alignant chaque caractère de x avec un gap et de même chaque caractère de y avec un gap.

Par exemple pour un mot x de longueur n et y de longueur n, on obtient cet alignement maximal suivant :

$$x_1x_2 \dots x_n - - \cdots -$$
$$- - \ \dots - y_1y_2 \dots y_n$$

2. Le problème d'alignement de séquences

2.1 Méthode naïve par énumération

Q.2.1.1 Etant donné $x \epsilon \sum^*$ un mot de longueur n, combien y a-t-il de mots $\bar{x}$ obtenus en ajoutant à x exactement k gaps ? Autrement dit combien y a-t-il de mots $\bar{x} \epsilon \sum^*$ tels que $|\bar{x}| = n + k \ \ et \, \pi(\bar{x}) = x$

Ce problème revient à trouver le nombre de manière de combiner k éléments parmi $n + k$ éléments, cela se traduit mathématiquement par :

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

Q.2.1.2 Etant donné un couple de mots (x, y) de longueur respective n et m avec $n \geq m$; une fois ajoutés k gaps à x pour obtenir un mot $\bar{x} \epsilon \sum^*$, combien de gaps seront ajoutés à y ?

On sait qu'après l'ajout des gaps sur chaque mot on a forcement $|\bar{x}| = |\bar{y}|$ donc :

$n + k = m + k_y \Leftrightarrow k_y = n + k - m$; le nombre de gaps à ajouter est donc $n + k - m$

Combien y a-t-il de façon d'insérer ces gaps dans y sachant qu'un gap du mot $\bar{y} \epsilon \sum^*$ ainsi obtenu ne doit pas être placé à la même position qu'un gap de $\bar{x}$? En déduire le nombre d'alignements possibles de (x, y).

Pour un mot $\bar{x}$ donné avec k_x gaps, le nombre de façon d'insérer k_y gaps dans y est égale à :

$$C_n^{k_y} = \binom{n}{n + k_x \ - m} = \frac{(n)!}{(n + k_x \ - m)!\,(m-k)!}$$

On obtient ainsi le nombre de façons d'insérer des gaps dans y de manière à respecter les contraintes d'alignement de $(\bar{x}, \bar{y})$ de la façon suivante :

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

On en déduit donc que le nombre d'alignements possibles de (x, y) est égale à :

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

Exemple avec : $|\bar{x}| = 15$ et $|\bar{y}| = 10$.

$$\sum_{k=0}^{10} \binom{15+k}{k} \times \binom{15}{5+k} = 298\ 199\ 265 \ \ \textit{alignements possibles}$$

Q.2.1.3 Quel genre de complexité temporelle aurait un algorithme naïf qui consisterait à énumérer tous les alignements de deux mots en vue de trouver la distance d'édition entre ces deux mots ? en vue de trouver un alignement de coût minimal ?

En vue de trouver la distance d'édition entre deux mots et de trouver un alignement de coût minimal, il faut dans un premier temps énumérer tous les alignements de mots, puis dans un second temps calculer leurs coûts respectifs afin de trouver le coût minimal. La complexité finale sera donc en $O(m \times (n+m)!)$.

Q.2.1.4 Quelle complexité spatiale aurait un algorithme naïf qui consisterait à énumérer tous les alignements de deux mots en vue de trouver la distance d'édition entres ces deux mots ? en vue de trouver un alignement de coût minimal ?

Soit le premier et le deuxième mot de taille respective $n \; et \; m$, il faut donc deux tableaux de taille respective n et m pour stocker les deux mots ce qui requiert une complexité spatiale de $O(n+m)$ et pour avoir un alignement de coût minimal, il faudrait avoir un tableau de taille max $n+m$ pour stocker l'alignement. Au total on obtient une complexité de $O(2(n+m))$.

<u>Tache A : Etude expérimentale</u>

- Résultat obtenu après le teste sur les instances **Inst_0000010_44.adn, Inst_0000010_7.adn et Inst_0000010_8.adn.**

```
Instance : <Inst_0000010_44.adn>
x = TATATGAGTC
y = TATTT
Distance d'edition : 10
Temps d execution : 0.0429 secondes

Instance : <Inst_0000010_7.adn>
x = TGGGTGCTAT
y = GGGGTTCTAT
Distance d'edition : 8
Temps d execution : 7.5441 secondes

Instance : <Inst_0000010_8.adn>
x = AACTGTCTTT
y = AACTGTTTT
Distance d'edition : 2
Temps d execution : 3.0918 secondes
```

- D'après les tests réalisés sur les instances, l'algorithme Naïf est capable d'exécuter 5 instances maximum en moins d'une minute.

- En utilisant également la commande top, nous avons observé que l'algorithme Naïf occupe 0.2% de la mémoire.

```
top - 00:28:40 up  4:56,  0 users,  load average: 0.15, 0.12, 0.16
Tasks:   9 total,   2 running,   7 sleeping,   0 stopped,   0 zombie
%Cpu(s): 12.5 us,  0.0 sy,  0.0 ni, 87.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   4632.5 total,   4496.7 free,     87.7 used,     48.1 buff/cache
MiB Swap:   2048.0 total,   2043.7 free,      4.3 used.   4404.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  406 root      20   0   15840   8956   5612 R 100.0   0.2   0:11.32 python3
  404 root      20   0   10872   3664   3152 R   0.3   0.1   0:00.06 top
    1 root      20   0     904      0      0 S   0.0   0.0   0:00.14 init
```

2.2 Programmation dynamique

a. Calcul de la distance d'édition par programmation dynamique

Q.2.2.1 Soit $(\bar{u} \cdot \bar{v})$ un alignement de $(x_{[1\dots i]}, y_{[1\dots j]})$ de longueur l.

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

Nous savons que par la définition d'un alignement il est impossible d'avoir $\bar{u}_l = -\ et\ \bar{v}_l = -$, donc si $\bar{u}_l = -$ alors $\bar{v}_l = y_j$ la dernière lettre de $y_{[1\dots j]}$

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

De même il est impossible d'avoir $\bar{v}_l = -\ et\ \bar{u}_l = -$, donc si $\bar{v}_l = -$ alors $\bar{u}_l = x_i$ la dernière lettre de $x_{[1\dots i]}$

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

$\bar{u}_l\ et\ \bar{v}_l$ sont les deux dernières lettres de l'alignement $(\bar{u}_l, \bar{v}_l)$, donc obligatoirement

$\bar{u}_l = x_i\ \ et\ \bar{v}_l = y_j$

Q.2.2.2 En distinguant les trois cas envisagés à la question précédente, exprimer $C(\bar{u} \cdot \bar{v})$ à partir de $C(\bar{u}_{[1\dots l-1]}, \bar{v}_{[1\dots l-1]})$.

$$\begin{cases} \text{Si } \bar{u}_l = - \text{ et } \bar{v}_l \neq -; \quad \text{alors } C(\bar{u}_l, \bar{v}_l) = c_{ins} + C(\bar{u}_{l-1}, \bar{v}_{l-1}) \\ \text{Si } \bar{u}_l \neq - \text{ et } \bar{v}_l = -; \quad \text{alors } C(\bar{u}_l, \bar{v}_l) = c_{del} + C(\bar{u}_{l-1}, \bar{v}_{l-1}) \\ \text{Si } \bar{u}_l \neq - \text{ et } \bar{v}_l \neq -; \quad \text{alors } C(\bar{u}_l, \bar{v}_l) = c_{sub} + C(\bar{u}_{l-1}, \bar{v}_{l-1}) \end{cases}$$

Q2.2.3 Pour $i \in [1 \dots n]\ et\ j \in [1 \dots m]$, en déduire l'expression de D (i, j) à partir des valeurs de D à des rangs plus petits

Pour calculer D (i, j) il faut prendre le minimum entre

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

Q2.2.4 Que vaut D (0, 0) ?

$D(0,0)$ correspond à la distance d'´édition entre deux sous mots vides, elle vaut donc 0

$$D(0,0) \;=\; 0$$

Q.2.2.5 Que vaut $D(0,j)$ pour j ϵ [1 … m] ?

i = 0, donc le sous mot $x[0...0]$ est vide, pour avoir un alignement partiel de longueur j, on aligne un gap avec chaque lettre $\epsilon\ y[1...j]$, on obtient que :

$$D(0,j) \;=\; \sum_{k=1}^{j} c_{ins} \;=\; j \times c_{ins}$$

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

j = 0, donc le sous mot $y[0...0]$ est vide, pour avoir un alignement partiel de longueur i, on aligne un gap avec chaque lettre $\epsilon\ x[1...j]$, on obtient que :

$$D(i,0) \;=\; \sum_{k=1}^{i} c_{del} \;=\; i \times c_{del}$$

Q.2.2.6 Donner le pseudo-code d'un algorithme itératif nommé DIST_1, qui prends en entrée deux mots, qui remplit un tableau à deux dimensions T avec toutes les valeurs de D pour finalement renvoyer la distance d'édition entre ces deux mots.

```
DIST_1
    Entrée : x et y deux mots,
    Sortie :  d(X,Y)
    n ← |x|
    m ← |y|
    T[0, 0] ← 0
    pour i = 1 à n faire
        T[i, 0] ← T[i - 1, 0] + c_del
    pour j = 1 à m faire
        T[0, j] ← T[0, j - 1] + c_ins
    pour i = 1 à n faire
        pour j = 1 à m faire
            d ← T[i - 1, j - 1] + c_sub(x[i], y[j])
            h ← T[i - 1, j] + c_del
            g ← T[i, j - 1] + c_ins
            T[i , j] ← min( d, h , g )
    Renvoie T[n, m]
```

Q.2.2.7 Quelle est la complexité spatiale de l'algorithme DIST_1 ?

On alloue un tableau a deux dimensions de taille $n \times m$ donc la complexité spatiale de DIST_1 est en $O(n \times m)$

Q.2.2.8 Quelle est la complexité temporelle de l'algorithme DIST_1 ?

On itère n fois la première boucle et m fois la deuxième boule et les opérations à l'intérieurs de ces boucles se font en temps constant, ensuite on itère sur deux boucles imbriquées, la première allant de 1 à n et la seconde de 1 à m. Les opérations à l'intérieure de la seconde boucle sont des comparaisons et des affectations, qui se font en temps constant. Donc au final on a une complexité en $O(n \times m)$.

b. Calcul de la distance d'édition par programmation dynamique

Q.2.2.8 Soit $(i, j) \epsilon [0 \dots n] \times [0 \dots m]$. Montrer que : Si $j > 0$ et $D(i,j) = D(i,j-1) + c_{ins}$, alors $\forall(\bar{s},\bar{t}) \epsilon Al^*(i,j-1), (\bar{s}\cdot -, \bar{t}\cdot y_j) \ \epsilon \ Al^*(i,j)$.

Supposons $D(i,j) = D(i,j-1) + c_{ins}$, avec $j > 0$

Montrons que $\forall(\bar{s},\bar{t}) \epsilon Al^*(i,j-1), (\bar{s}\cdot -, \bar{t}\cdot y_j) \epsilon Al^*(i,j)$:

Soit $\forall(\bar{s},\bar{t}) \ \epsilon \ Al^*(i,j-1)$ on a :

$C(\bar{s},\bar{t}) = d\left(x_{[1\dots i]}, y_{[1\dots j-1]}\right)$

Or, $C\left(\bar{s}\cdot -, \bar{t}\cdot y_j\right) = C(\bar{s},\bar{t}) + c_{ins} = D(i,j-1) + c_{ins} = D(i,j)$ par hypothèse

Q.2.2.10 Donner le pseudo-code d'un algorithme itératif nommé SOL_1, qui à partir d'un couple de mots (x, y) et d'un tableau T indexé par $[0 \dots |x|] \times [0 \dots |y|]$ contenant les valeurs de D, renvoie un alignement minimal de (x, y).

```
SOL_1
    Entrée : x et y deux mots,
    T : tableau indexé par [0...|x|]x[0...|y|] contenant les valeurs de D
    Sortie :  Un alignement minimal de (x,y)
    i ← |x|
    j ← |y|
    x'← "" #alignement de x
    y'← "" #alignement de y

    Tant que i > 0 et j > 0 faire :
        val_case = T[i,j]
        d ← T[i - 1, j - 1]
        h ← T[i - 1, j]
        g ← T[i, j - 1]
        si (val_case – d) est égale à c_sub(x[i], y[i]) Alors :
            x'← x[i].x'
            y'← y[i].y'
            i ← i - 1
            j ← j - 1
        sinon si (val_case – h) est égale à c_del Alors :
            x'← x[i].x'
            y'← "-".y'
            i ← i - 1
        sinon si (val_case - g) est égale à  c_ins Alors :
            x'← "-".x'
            y'← y[j].y'
            j ← j - 1
    Tant que i > 0 faire :
        x'← "-".x'
        i ← i - 1
    Tant que j > 0 faire :
        y'← "-".y'
        j ← j - 1
    Retourner (x', y')
```

Q.2.2.11 En combinant les algorithmes DIST_1 et SOL_1 avec quelle complexité temporelle résout-on le problème ALI ?

SOL_1 fait au plus $n + m$ iterations et toutes opérations d'affectations et de comparaisons se font en temps constant donc SOL_1 est en $O(n + m)$, on a calculé précédemment que DIST_1 était en $O(n \times m)$ donc en combinant les deux algorithmes DIST_1 et SOL_1, on obtient une complexité temporelle en $O((n \times m) + n + m)$ = $O(n \times m)$

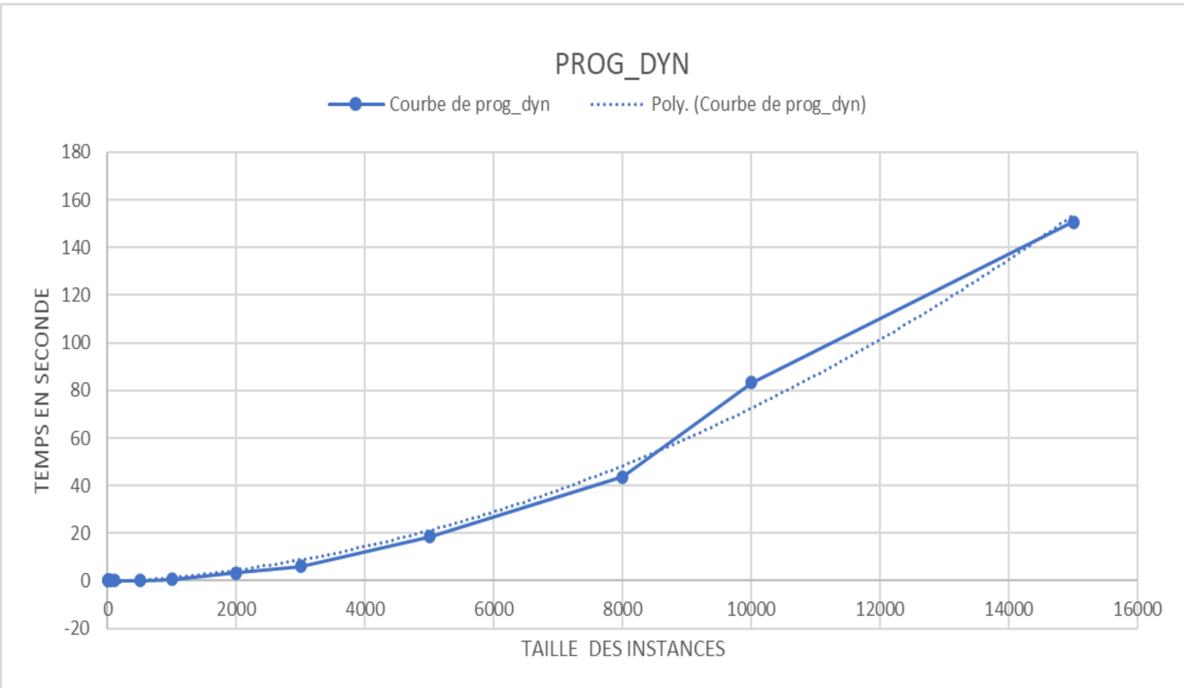
Q.2.2.12 En combinant les algorithmes DIST_1 et SOL_1 avec quelle complexité spatiale résout-on le problème ALI ?

En combinant les deux algorithmes DIST_1 et SOL_1, on obtient une complexité spatiale en $O((n \times m) + n + m) \approx O(n \times m)$, car on a besoin d'une matrice $n \times m$ pour stocker les coûts de chaque alignement (les valeurs de D) et de deux chaines de caractères de taille maximale $n + m$ pour renvoyer l'alignement optimal.

Tache B : Etude expérimentale

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

L'allure de la courbe obtenue est polynomiale donc l'étude expérimentale correspond bien au résultat théorique.



Nous avons essayé d'estimer la consommation en mémoire d'une instance de taille 20.000, le programme a craché à la fin car PROG_DYN occupait beaucoup la mémoire, à peu près 95% de la mémoire.

```
top - 00:12:41 up  4:40,  0 users,  load average: 1.21, 0.64, 0.27
Tasks:   9 total,   2 running,   7 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3.4 us,  2.3 sy,  0.0 ni, 91.9 id,  2.4 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   4632.5 total,     88.5 free,   4520.4 used,     23.7 buff/cache
MiB Swap:   2048.0 total,      0.0 free,   2048.0 used.     17.7 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  397 root      20   0 6643948   4.3g   1080 R  46.0  95.7   2:52.79 python3
  390 root      20   0   10872    268      0 R   4.2   0.0   0:07.61 top
    1 root      20   0     904      0      0 S   0.0   0.0   0:00.14 init
  191 root      20   0     904      4      0 S   0.0   0.0   0:00.02 init
  192 root      20   0     904      0      0 S   0.0   0.0   0:00.39 init
  193 root      20   0   10172      4      4 S   0.0   0.0   0:03.05 bash
  368 root      20   0     904      0      0 S   0.0   0.0   0:00.00 init
  369 root      20   0     904     12      0 S   0.0   0.0   0:00.29 init
  370 root      20   0   10040      8      4 S   0.0   0.0   0:00.24 bash
```

2.3 Amélioration de la complexité spatiale du calcul de la distance

Q.2.3.1 Expliquer pourquoi lors du remplissage de la ligne i > 0 du Tableau T dans l'algorithme DIST_1, il suffirait d'avoir accès aux lignes i − 1 et i du tableau (partiellement rempli pour cette dernière).

Pour calculer la case $T[\mathrm{i}, \mathrm{j}]$ du tableau dans DIST_1, on a juste besoin de connaitre les cases $T[i, j-1]$, $T[i-1, j]$ et $T[i-1, j-1]$. Donc pour calculer la ligne i de la matrice T, on a besoin que de la ligne $i-1$. Donc Il suffit de maintenir 2 lignes de la matrice T.

Q.2.3.2 Donner le pseudo-code d'un algorithme itératif DIST_2, qui a la même spécification que DIST_1, mais qui a une complexité spatiale en O(m).

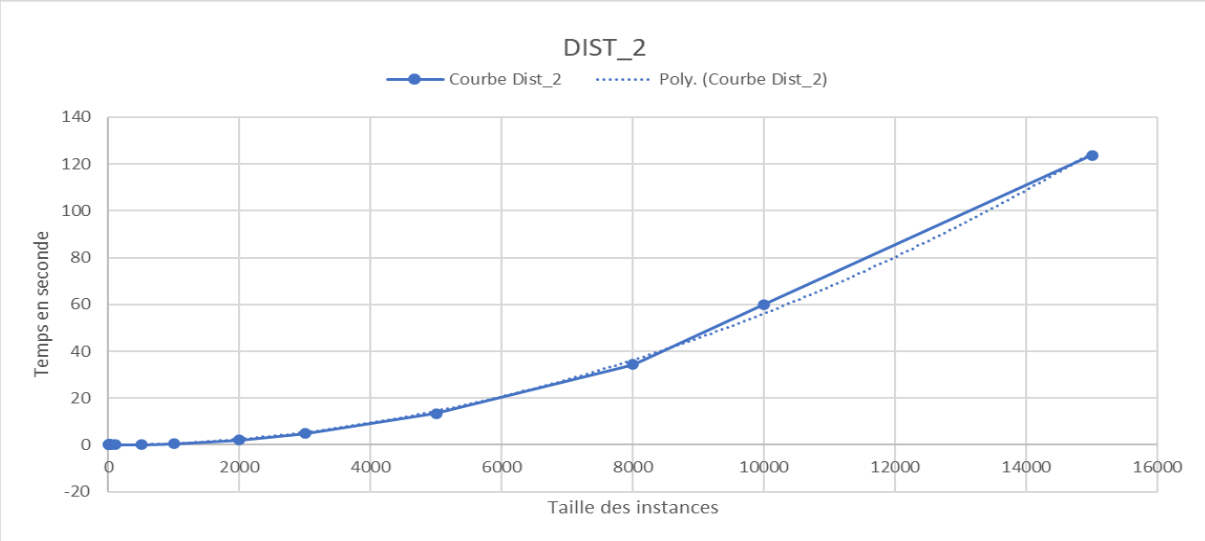
```
DIST_2
    Entrée : x et y deux mots,
    Sortie :  d(X,Y)
    n ← |x|
    m ← |y|
    Tab_1[0...m] #pour ligne 1
    Tab_2[0...m] #pour ligne 2

    Tab_1[0] ← 0
    pour j = 1 à m faire
        Tab_1[j] ← Tab_1[j - 1] + c_ins

    pour i = 1 à n faire
        Tab_2[0] ← Tab_1[0] + c_del
        pour j = 1 à m faire
            d ← Tab_1[j - 1] + csub(x[i], y[j])
            h ← Tab_1[j] + c_del
            g ← Tab_2[j - 1] + cins
            Tab_2[j] ← min( d, h , g )
        Tab_1 , Tab_2 = Tab_2, Tab_1 #echanger Tab_1 et Tab_2
    Renvoie Tab_1[m]
```

Tache C : Etude expérimentale

- Courbe de consommation de temps CPU en fonction de la taille |x| du premier mot du couple des instances fournies.



On voit que sur la figure ci-dessus que la courbe de DIST_2 a la même allure que celle d'une courbe polynomiale donc le résultat théorique correspond bien à l'étude expérimentale.

DIST_1 et DIST_2 ont la même complexité temporelle, mais l'avantage de DIST_2 c'est qu'elle consomme beaucoup moins de mémoire que DIST_1.

La quantité de mémoire utilisé par DIST_2 est 0.2% de la mémoire, le teste a été réalisé sur une instance de taille 20.000

```
top - 00:21:10 up  4:49,  0 users,  load average: 0.66, 0.33, 0.23
Tasks:   9 total,   2 running,   7 sleeping,   0 stopped,   0 zombie
%Cpu(s): 12.5 us,  0.0 sy,  0.0 ni, 87.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   4632.5 total,   4513.6 free,     88.1 used,     30.7 buff/cache
MiB Swap:   2048.0 total,   2043.7 free,      4.3 used.   4412.8 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  401 root      20   0   17212  10188   5648 R 100.0   0.2   1:08.10 python3
    1 root      20   0     904      0      0 S   0.0   0.0   0:00.14 init
  191 root      20   0     904      4      0 S   0.0   0.0   0:00.02 init
  192 root      20   0     904      0      0 S   0.0   0.0   0:00.51 init
```

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

Q.2.4.1 Donner le pseudo code d'une fonction mot_gaps qui, étant donné un entier naturel k, renvoie le mot constitué de k gaps.

```
mot_gaps:
    Entrée : k un entier
    Sortie :  un mot de k gaps
    T ← ""
    pour i = 1 a k faire:
        T ← "-".T
    Retourner T
```

Q.2.4.2 Donner le pseudo code d'une fonction align_lettre_mot qui, étant donné x un mot de longueur 1 et y un mot non vide de longueur quelconque, renvoie un meilleur alignement de (x, y).

```
align_lettre_mot :
    Entrée : x un mot de longueur 1 et y un mot non vide
    Sortie : un meilleur alignement de (x, y)

    nb_gaps ← 0
    cout_min ← c_sub(x, y[0])
    m ← |y|

    pour i = 1 a m faire:
        si cout_min > c_sub(x, y[i]) :
            cout_min ← c_sub(x, y[i])
            nb_gaps ← i

    Retourner mots_gap(nb_gaps) + x + mots_gap(m-nb_gaps-1)
```

Q.2.4.3 On considère un exemple où $\sum$ est l'alphabet latin, constitué de 26 lettres majuscules, x = BALLON et y = ROND. On coupe x et y en leur milieux : x^1 = BAL, x^2 = LON, y^1 = RO et y^2 = ND.

$$\text{On fixe } c_{ins} = c_{del} = 3, \text{ et } c_{sub}(a,b) = \begin{cases} 0 \text{ si } a = b \\ 5 \text{ si } a \text{ et } b \text{ sont deux voyelles distinctes} \\ 5 \text{ si } a \text{ et } b \text{ sont deux consonnes distinctes} \\ 7 \text{ sinon} \end{cases}$$

Donner $(\bar{s}, \bar{t})$ un alignement optimal de (x^1, y^1) et $(\bar{u}, \bar{v})$ un alignement optimal de (x^2, y^2). Montrer que $(\bar{s} \cdot \bar{u}, \bar{t} \cdot \bar{v})$ n'est pas un alignement optimal de $(x,\ y)$.

$(\bar{s}, \bar{t})$ est un alignement optimal de (x^1, y^1) avec :

$$\begin{array}{l} \bar{s}: \mathrm{B\ A\ L} \\ \bar{t}: \mathrm{R\ O\ -} \end{array}$$: le coût est 13

$(\bar{u}, \bar{v})$ est un alignement optimal de (x^2, y^2) avec :

$$\begin{array}{l} \bar{u}: \mathrm{L\ O\ N\ -} \\ \bar{v}: \mathrm{-\ -\ N\ D} \end{array}$$: le coût est 9

$(\bar{s} \cdot \bar{u}, \bar{t} \cdot \bar{v})$ n'est pas un alignement optimal de $(x,\ y)$ car il existe un meilleur alignement :

$$\begin{array}{l} \bar{s}\cdot\bar{u}: \mathrm{B\ A\ L\ L\ O\ N\ -} \\ \bar{t}\cdot\bar{v}: \mathrm{-\ -\ -\ R\ O\ N\ D} \end{array}$$: le coût est 17

Etant donné que le coût de $(\bar{s} \cdot \bar{u}, \bar{t} \cdot \bar{v})$ vaut la somme des coûts de $(\bar{s}, \bar{t})$ *et* $(\bar{u}, \bar{v})$ qui est égale à 13 + 9 = 21 et qu'on a 21 > 17, on peut donc conclure que $(\bar{s} \cdot \bar{u}, \bar{t} \cdot \bar{v})$ n'est pas un alignement optimal de $(x,\ y)$.

Q.2.4.4 En supposant disposer de la fonction coupure, donner le pseudo code de l'algorithme récursif de type diviser pour régner, nommé SOL_2, qui à partir d'un couple de mots (x, y) calcule un alignement minimal de (x, y).

```
SOL_2 :
    Entrée : x et y deux mot
    Sortie : un alignement minimal de (x, y).

    n ← |x|
    m ← |y|

    si n == 0 :
        Renvoyer mots_gap(m), y
    si m == 0 :
        Renvoyer x, mots_gap(n)
    si n == 1 :
        Renvoyer align_lettre_mot(x,y), y
    si m == 1 :
        Renvoyer x, align_lettre_mot(y, x)
    mid ← math.floor(n/2)
    cut_y ← coupure(x, y)

    s, t ← SOL_2(x[:mid],y[:cut_y])
    u, v ← SOL_2(x[mid:],y[cut_y:])

    Renvoyre (s+u, t+v)
```

Q.2.4.5 Donner le pseudo-code d'une fonction coupure telle que décrite ci-dessus.

```
COUPURE :
   Entrée : x et y deux mots
   Sortie : un alignement minimal
   n ← |x|
   m ← |y|

   T = [[], []]  # Stock les distances
   col = [[], []]  # Stock la colonne d'origine sur la ligne i

   pour i = 0 à m faire :
       (T[0]).append(j * c_ins)  # Initialisation
       (T[1]).append(-1)  # Remplissage

       (col[0]).append(j)  # Initialisation
       (col[1]).append(-1)  # Remplissage

   pour i = 1 à n/2 faire :
       T[1][0] ← i * c_del
       pour j = 1 à m + 1 faire :
           T[1][j] ← min([T[1][j - 1] + c_ins,
                               T[0][j] + c_del,
                               T[0][j - 1] + c_sub(x[i - 1], y[j - 1])])
       T[0] = [c for c in T[1]]

   pour i = (n/2)+1 à n faire :
       T[1][0] ← i * c_del
       pour j = 1 à m faire :
           val1 ← T[1][j - 1] + c_ins
           val2 ← T[0][j] + c_del
           val3 ← T[0][j - 1] + c_sub(x[i - 1], y[j - 1])
           T[1][j] ← min(val1, val2, val3)

           Si T[1][j] est égale val1:
               col[1][j] ← col[1][j - 1]
           if T[1][j] est égale val2:
               col[1][j] ← col[0][j]
           if T[1][j] est égale val3:
               col[1][j] ← col[0][j - 1]

       T[0] = [c for c in T[1]]
       col[0] = [c for c in col[1]]

   Retourner col[0][m]
```

Q.2.4.6 Quelle est la complexité spatiale de coupure ?

On utilise deux tableaux, de deux lignes chacun. Le premier stock les distances, tandis que le deuxième stock la colonne d'origine. On a donc $O(m) + O(m)$, ce qui donne une complexité spatiale en $O(m)$.

Q.2.4.7 Quelle est la complexité spatiale de SOL_2 ?

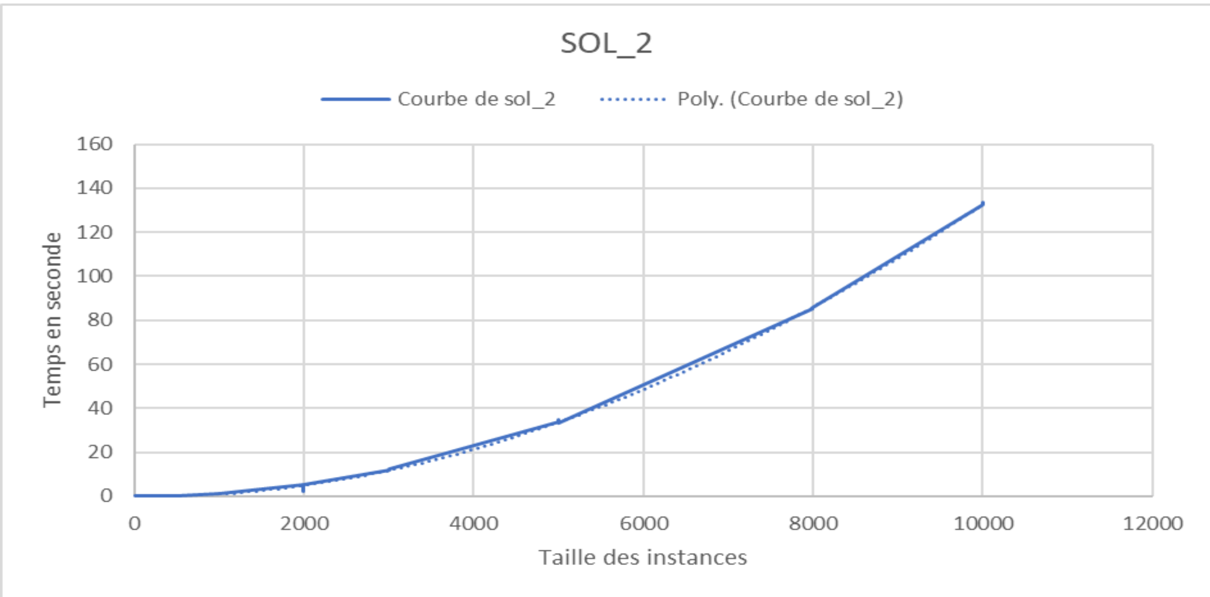
La complexité spatiale de SOL_2 est en $O(n+m)$ car chaque appel récursif prend un espace de $O(m)$ pour manipuler 2 séquences (gauche et droite) et on a besoin de $O(n+m)$ pour stocker les deux séquences.

Q.2.4.8 Quelle est la complexité temporelle de coupure ?

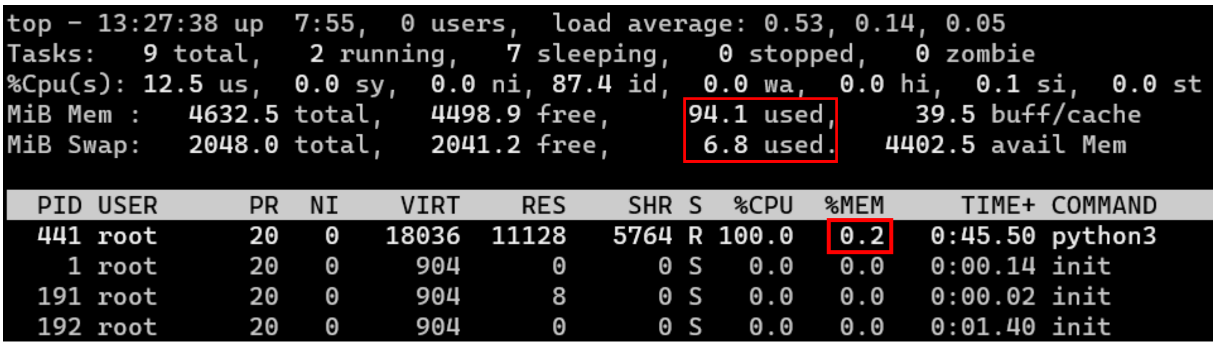
On parcourt une boucle de m+1 itérations, elle a donc une complexité en $O(m)$. Il reste ensuite deux boucles à parcourir. La première est en $O(n/2)$ car n/2 itérations. La dernière possède aussi n/2 itérations, et contient une boucle imbriquée qui possède m+1 itérations. La dernière boucle est donc en $O((n/2) \times m)$. La complexité de la fonction coupure est donc en $O(n \times m)$.

Tâche D : Etude expérimentale

Courbe de consommation de temps CPU en fonction de la taille |x| du premier mot du couple des instances fournies.



Avec SOL_2, il y'a juste 0.2% de la mémoire qui est consommé.



```
top - 13:27:38 up  7:55,  0 users,  load average: 0.53, 0.14, 0.05
Tasks:   9 total,   2 running,   7 sleeping,   0 stopped,   0 zombie
%Cpu(s): 12.5 us,  0.0 sy,  0.0 ni, 87.4 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0 st
MiB Mem :   4632.5 total,   4498.9 free,     94.1 used,     39.5 buff/cache
MiB Swap:   2048.0 total,   2041.2 free,      6.8 used.   4402.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  441 root      20   0   18036  11128   5764 R 100.0   0.2   0:45.50 python3
    1 root      20   0     904      0      0 S   0.0   0.0   0:00.14 init
  191 root      20   0     904      8      0 S   0.0   0.0   0:00.02 init
  192 root      20   0     904      0      0 S   0.0   0.0   0:01.40 init
```

Q.2.4.9 A-t-on perdu en complexité temporelle en améliorant la complexité spatiale ? Comparez expérimentalement la complexité temporelle de SOL_2 à celle de SOL_1.

En améliorant la complexité spatiale de SOL_2 on a perdu en complexité temporelle.

- SOL_1 a une complexité temporelle de $O(n + m)$ et une complexité spatiale de $O(n \times m)$
- SOL_2 a une complexité temporelle de $O(n \times m)$ et une complexité spatiale de $O(n + m)$