



Projet algo

Tags	INFO
Code	IN003

PROJET : ALIGNEMENT DE SÉQUENCES

PROJET : ALIGNEMENT DE SÉQUENCES

2. PROBLÈME D'ALIGNEMENT DE SÉQUENCES

[Question 1](#)

[Question 2](#)

3. ALGORITHMES POUR L'ALIGNEMENT DE SÉQUENCES

[Question 3](#)

[Question 4](#)

[Question 5](#)

[Question 6](#)

[Tâche A](#)

[Question 7](#)

[Question 8](#)

[Question 9](#)

[Question 10](#)

[Question 11](#)

[Question 12](#)

[Question 13](#)

[Question 14](#)

[Question 15](#)

[Question 16](#)

[Question 17](#)

[Question 18](#)

[Tâche B](#)

[Question 19](#)

[Question 20](#)

[Tâche C](#)

[Question 21](#)

[Question 22](#)

[Question 23](#)

[Question 24](#)

[Question 25](#)

[Question 26](#)

[Question 27](#)

[Question 28](#)

[Tâche D](#)

[Question 29](#)

2. PROBLÈME D'ALIGNEMENT DE SÉQUENCES

▼ Question 1

Pour 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 $(x \cdot u, y \cdot v)$, on montre les quatre points de la définition 2.2 :

- $\pi(\bar{x}.\bar{u}) = \pi(\bar{x}).\pi(\bar{u}) = x.u$
- $\pi(\bar{y}.\bar{v}) = \pi(\bar{y}).\pi(\bar{v}) = y.v$
- $|\bar{x}.\bar{u}| = |\bar{x}| + |\bar{u}| = |\bar{y}| + |\bar{v}| = |\bar{y}.\bar{v}|$
- si $i > |\bar{x}| = |\bar{y}|$ alors $(\bar{x}, \bar{u})_i = \bar{u}_i \neq -$ ou $(\bar{y}, \bar{v})_i = \bar{v}_i \neq -$
si $i \leq |\bar{x}| = |\bar{y}|$ alors $(\bar{x}, \bar{u})_i = \bar{x}_i \neq -$ ou $(\bar{y}, \bar{v})_i = \bar{y}_i \neq -$

▼ Question 2

Si $x \in \Sigma^$ est un mot de longueur n et $y \in \Sigma^*$ est un mot de longueur m , quelle est la longueur maximale d'un alignement de (x, y) ?*

La longueur maximale d'un alignement de (x, y) serait de $n + m$

ex : Soit $x = x_1x_2...x_n$ et $y = y_1y_2...y_m$

On définit l'alignement suivant:

$$\begin{aligned}\bar{x} : & x_1 \ x_2 \ \dots \ x_n \ -1 \ -2 \ \dots \ -m \\ \bar{y} : & -1 \ -2 \ \dots \ -n \ \ y_1 \ y_2 \ \dots \ y_m\end{aligned}$$

Tel que $|\bar{x}| = n + m$

3. ALGORITHMES POUR L'ALIGNEMENT DE SÉQUENCES

3.1 MÉTHODE NAÏVE PAR ÉNUMÉRATION

▼ Question 3

Étant donné $x \in \Sigma^*$ un mot de longueur n , on regarde le nombre de façon d'insérer k gaps dans $n + k$ emplacements soit $C_{n+k}^k = \binom{n+k}{k} = \frac{(n+k)!}{k!(n+k-k)!} = \frac{(n+k)!}{k! n!}$ mots \bar{x} .

▼ Question 4

- On ajoute k gaps à \bar{x} , qui est de longueur $n \geq m$, d'où $|\bar{x}| = k + n$. On cherche alors le nombre k' de gaps que l'on rajoute à y :

$$\begin{aligned}|\bar{y}| = m + k &\implies m + k' = k + n \text{ car } |\bar{x}| = |\bar{y}| \\ &\implies k' = k + n - m\end{aligned}$$

- À k fixé tel que $|\bar{x}| = k + n$, on regarde le nombre de façons d'insérer k' gaps parmi n emplacements qui correspondent aux \bar{x}_i qui ne sont pas des gaps, soit exactement n emplacements.

Ainsi il y a $C_n^{k'} = \binom{n}{k'} = \binom{n}{k+n-m} = \frac{n!}{(k+n-m)!(m-k)!}$ façons d'insérer ces gaps dans y .

- Pour avoir le nombre d'alignements de (x, y) possible, on regarde le nombre de mots \bar{x} possibles, soit pour $k \in \{0, \dots, m\}$, $\binom{n+k}{k}$ puis le nombre de mots \bar{y} possibles qui préservent la validité de l'alignement, soit $\binom{n}{k+n-m}$ choix.

Ainsi, pour tout $0 \leq k \leq m$ le nombre d'alignements est $\binom{n+k}{k} \binom{n}{n+k-m}$.

Soit enfin, un nombre total d'alignements possibles de :

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

→ Exemple : Pour $|x| = 15$ et $|y| = 10$ on obtient:

$$\sum_{k=0}^{10} \binom{15+k}{k} \binom{15}{5+k} = 298\,199\,265 \text{ combinaisons possibles}$$

▼ Question 5

Au maximum, chacun des deux tableaux est de taille $n + m$.

Un tableau de taille $n + m$ admet $(n + m)!$ permutations par définition des permutations d'un ensemble.

Comme $m \leq n$, on a que l'algorithme naïf qui liste toutes les possibilités est en $O(2(n + m)!) \equiv O(2(2n)!) \equiv O(n!)$

Un algorithme naïf trouvant la distance d'édition ou un alignement de coût minimal et donc de complexité temporelle factorielle.

▼ Question 6

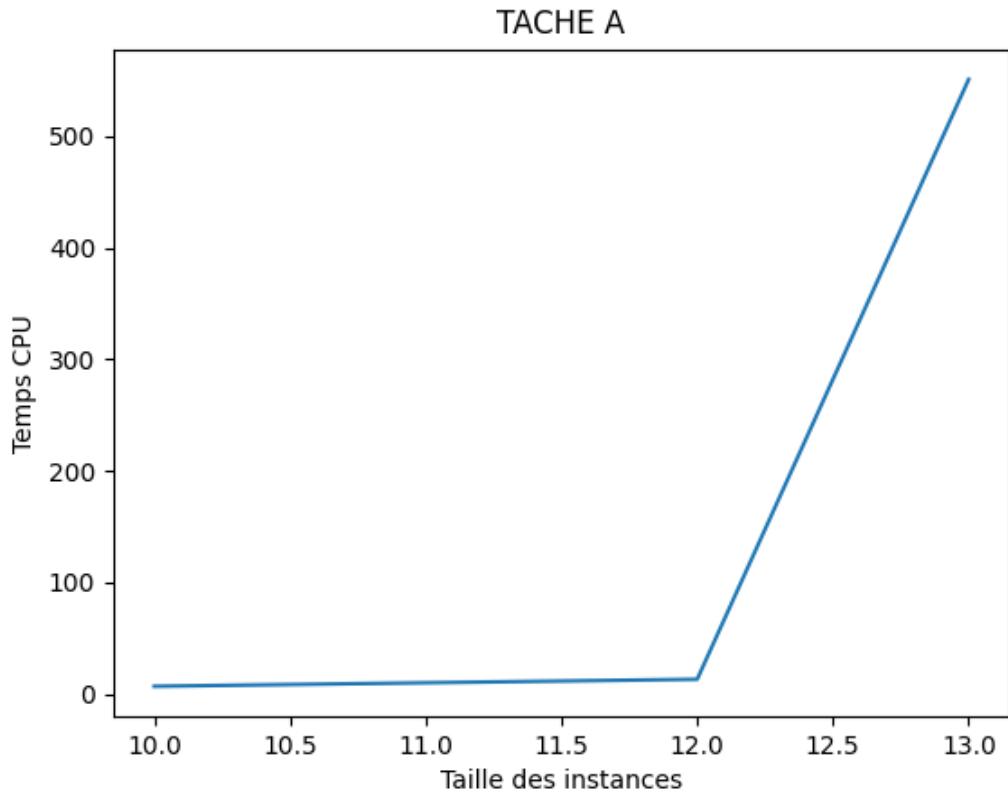
Un algorithme naïf qui consisterait à énumérer tous les alignements de deux mots en vue de trouver la distance d'édition entre ces deux mots stockerait en mémoire chacun de ces 2 tableaux générés, de taille maximale $n + m$, et est donc de complexité spatiale $2(n + m) * O(n!) \equiv O(n!)$.

De même, en vue de trouver un alignement de coût minimal on aurait une complexité spatiale de $O(n!)$.

▼ Tâche A

Cette représentation graphique du temps d'exécution de `DIST_NAIF` en fonction de la taille des instances nous montre qu'à partir d'instances de taille 13, la résolution requiert plus d'une minute (La dernière dont l'exécution prenant moins d'une minute est `Inst_0000012_56.adn`)

On vérifie bien sur ces instances que la complexité spatiale de `DIST_NAIF` est exponentielle



3.2 PROGRAMMATION DYNAMIQUE

▼ Question 7

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

- Si $\bar{u}_l = -$, $\bar{v}_l \neq -$, c'est-à-dire \bar{v}_l est une lettre: $\bar{v}_l = y_j$
- Si $\bar{v}_l = -$, $\bar{u}_l \neq -$, c'est-à-dire \bar{u}_l est une lettre: $\bar{u}_l = x_i$
- Si $\bar{u}_l \neq -$ et $\bar{v}_l \neq -$, alors \bar{u}_l et \bar{v}_l sont des lettres : $\bar{u}_l = x_i$ et $\bar{v}_l = y_j$

▼ Question 8

Exprimer $C(\bar{u}, \bar{v})$ à partir de $C(\bar{u}_{[1..l-1]}, \bar{v}_{[1..l-1]})$:

- Si $\bar{u}_l = -$, alors $C(\bar{u}, \bar{v}) = C(\bar{u}_{[1..l-1]}, \bar{v}_{[1..l-1]}) + c_{ins}$
- Si $\bar{v}_l = -$, alors $C(\bar{u}, \bar{v}) = C(\bar{u}_{[1..l-1]}, \bar{v}_{[1..l-1]}) + c_{del}$
- Si $\bar{u}_l \neq -$ et $\bar{v}_l \neq -$, alors $C(\bar{u}, \bar{v}) = C(\bar{u}_{[1..l-1]}, \bar{v}_{[1..l-1]}) + c_{sub}(\bar{u}_l, \bar{v}_l)$

▼ Question 9

Pour $i \in [1..n]$ et $j \in [1..m]$, $D(i, j) = d(x_{[1..i]}, y_{[1..j]})$

$D(i, j)$ peut s'exprimer en fonction des trois sous-problèmes plus petits possibles que sont $D(i', j')$, $D(i, j')$ et $D(i', j)$ de telles sortes:

- $D(i', j) + c_{ins}$
- $D(i, j') + c_{del}$
- $D(i', j') + c_{sub}(i, j)$

Par définition, la distance d'édition à laquelle correspond D est le minimum du coût de l'alignement à tout moment, ainsi $D(i, j)$ est égal à la plus petite des 3 valeurs possibles:

$$D(i, j) = \min\{D(i', j') + c_{sub}(i, j), D(i, j') + c_{del}, D(i', j) + c_{ins}\}$$

▼ Question 10

Pour le cas de base $i = 0$ et $j = 0$, on aura $D(0, 0) = d(x_{[1,0]}, y_{[1,0]}) = d(\varepsilon, \varepsilon) = 0$, car la distance d'édition entre deux mots vides est nulle.

▼ Question 11

Pour les deux autres cas de base :

- Pour $i \in \llbracket 1, n \rrbracket$, on a $D(i, 0) = i \times c_{del} = 2i$ qui correspond au cas où on doit réaliser i délétions.
- Pour $j \in \llbracket 1, m \rrbracket$, on a $D(0, j) = j \times c_{ins} = 2j$ qui correspond au cas où on doit réaliser j insertions.

▼ Question 12

Algorithme DIST_1:

Entrée: Deux mots x et y de taille n et m tels que $(x, y) \in \Sigma^* \times \Sigma^*$
Sortie: La distance d'édition entre x et y tout en remplissant le tableau T
qui est de dimension $n*m$

```

pour i ∈ [0, ..., n] faire:
    T[i][0] = i*c_del
fin pour
pour j ∈ [1, ..., m] faire:
    T[0][j] = j*cins
fin pour

pour i ∈ [1, ..., n] faire:
    pour j ∈ [1, ..., m] faire:
        T[i][j]=min(T[i-1][j]+cdel,T[i][j -1]+cins,T[i-1][j-1]+csub(x[i-1],y[j-1]))
    fin pour
fin pour

```

```
Retourner T[n][m]
```

▼ Question 13

On construit un tableau de n lignes et m colonnes, donc de taille $n \times m$ donc la complexité spatiale de `DIST_1` est en $\Theta(nm)$.

Comme on a $m \leq n$, l'algorithme est alors en $\Theta(n^2)$, soit en espace polynomial.

$(\Theta(n^2) \times 16$ bits car chaque case contient un entier, et est donc encodée sur 16 bits).

▼ Question 14

Comme chaque calcul se fait en $\Theta(1)$ (temps constant) la complexité temporelle de `DIST_1` est en $\Theta((n + 1) + (m + 1) + nm) \equiv \Theta(nm)$.

Comme on a $m \leq n$, l'algorithme est alors en $\Theta(n^2)$, soit en temps polynomial.

▼ Question 15

$$(i, j) \in [1..n] \times [1..m]$$

On suppose $j > 0$ et $D(i, j) = D(i, j - 1) + c_{ins}$.

Soit $(\bar{s}, \bar{t}) \in Al^*(i, j - 1)$, montrons que $(\bar{s}.-, \bar{t}.y_j) \in Al^*(i, j)$ i.e. $(\bar{s}.-, \bar{t}.y_j)$ est un alignement de $(x_{[1..i]}, y_{[1..j]})$ et $C(\bar{s}.-, \bar{t}.y_j) = d(x_{[1..i]}, y_{[1..j]})$.

- $\pi(\bar{s}.-) = \pi(\bar{s}) = x_{[1..i]}$ et $\pi(\bar{t}.y_j) = \pi(\bar{t}) + \pi(y_j) = y_{[1..j-1]} \cdot y_j = y_{[1..j]}$ donc $(\bar{s}.-, \bar{t}.y_j) \in Al(i, j)$.
- $C(\bar{s}.-, \bar{t}.y_j) = C(\bar{s}, \bar{t}) + c(-, y_j) = d(x_{[1..i]}, y_{[1..j-1]}) + c_{ins} = D(i, j - 1) + c_{ins} = D(i, j) = d(x_{[1..i]}, y_{[1..j]})$

Donc $(\bar{s}.-, \bar{t}.y_j) \in Al^*(i, j)$.

▼ Question 16

```
Algorithme SOL_1:
```

```
Entrée: deux mots x et y de taille n et m tels que (x, y) ∈ Σ* × Σ* et
        un tableau T indexé par [0, ..., n] × [0, ..., m] contenant les valeurs de D
Sortie: Alignement minimal de (x, y)
```

```
u = []
```

```

v = []
i = n
j = m

tant que i > 0 et j > 0 faire:
    si T[i][j] = T[i][j-1]+cins faire:
        insérer - en tête de u
        insérer y[j-1] en tête de v
        j = j - 1
        passer à l'itération suivante
    fin si
    si T[i][j] = T[i-1][j]+cdel alors:
        insérer x[i-1] en tête de u
        insérer - en tête de v
        i = i - 1
        passer à l'itération suivante
    fin si
    si T[i][j] = T[i-1][j-1]+csub(x[i-1],y[j-1]) alors:
        insérer x[i-1] en tête de u
        insérer substitution(yi-1,xi-1) soit x[i-1] en tête de v
        i = i - 1
        j = j - 1
        passer à l'itération suivante
    fin si
fin tant que

tant que i > 0 faire:
    insérer x[i-1] en tête de u
    insérer - en tête de v
    i = i - 1
fin tant que

tant que j > 0 faire:
    insérer - en tête de u
    insérer y[j-1] en tête de v
    j = j - 1
fin tant que

Retourner u , v

```

▼ Question 17

L'algorithme `DIST_1` remplit le tableau T en $\Theta(nm)$ et l'algorithme `SOL_1` effectue la recherche de l'alignement optimal en $\Theta(n + m)$.

Ainsi, en combinant les algorithmes `DIST_1` et `SOL_1`, on résout le problème *ALI* avec une complexité temporelle de $\Theta(n + m + nm) \equiv \Theta(nm)$

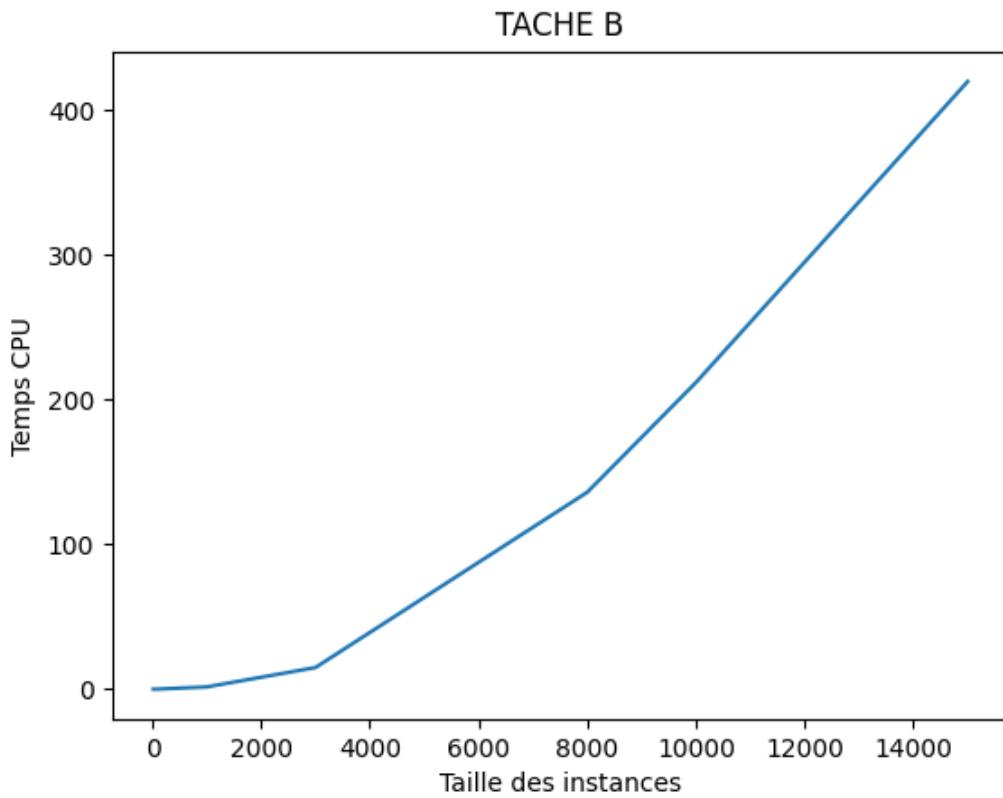
▼ Question 18

L'algorithme `SOL_1` se contente de parcourir à l'envers le tableau retourné par `DIST_1`, et ne modifie donc pas sa complexité spatiale. Ainsi en combinant les

algorithmes `DIST_1` et `SOL_1`, on résout le problème *ALI* avec une complexité spatiale de $\Theta(nm)$ (x16 bits car chaque case contient un entier)

▼ Tâche B

Cette représentation graphique du temps d'exécution de `PROG_DYN` en fonction de la taille des instances corrobore bien la complexité théorique trouvée précédemment (courbe d'une fonction polynomiale).



Nous avons ensuite estimé la consommation mémoire sur une instance de grande taille *Inst_0015000.adn* avec $|x| = 15000$ et $|y| = 13395$ et en considérant que les caractères sont encodés sur 8 bits. On peut donc calculer que l'on aura $15000 * 13395 * 8 = 1,6 \cdot 10^9$ bits utilisés.

3.3 AMÉLIORATION DE LA COMPLEXITÉ SPATIALE DU CALCUL DE LA DISTANCE

▼ Question 19

Nous avons vu dans l'algorithme `DIST_1` (q.12) qu'afin de remplir la case $T[i][j]$, les cases dont nous avons besoin pour le calcul sont les seules cases $T[i - 1][j - 1]$, $T[i - 1][j]$ et $T[i][j - 1]$.

Ainsi, une fois les valeurs des la ligne $i - 1$ et i calculées, toutes les lignes antérieures dans le tableau ne sont plus utiles pour le calcul de la distance d'édition.

▼ Question 20

Algorithme `DIST_2`:

```

Entrée: Deux mots x et y de taille n et m tels que (x, y) ∈ Σ* × Σ*
Sortie: La distance d'édition entre x et y tout en remplaissant le tableau T
        qui est de dimension 2*m

cpt = 0
i = 1
pour j ∈ [0, ..., m] faire:
    T[0][j] = j*cins
fin pour
T [1][0] = cdel

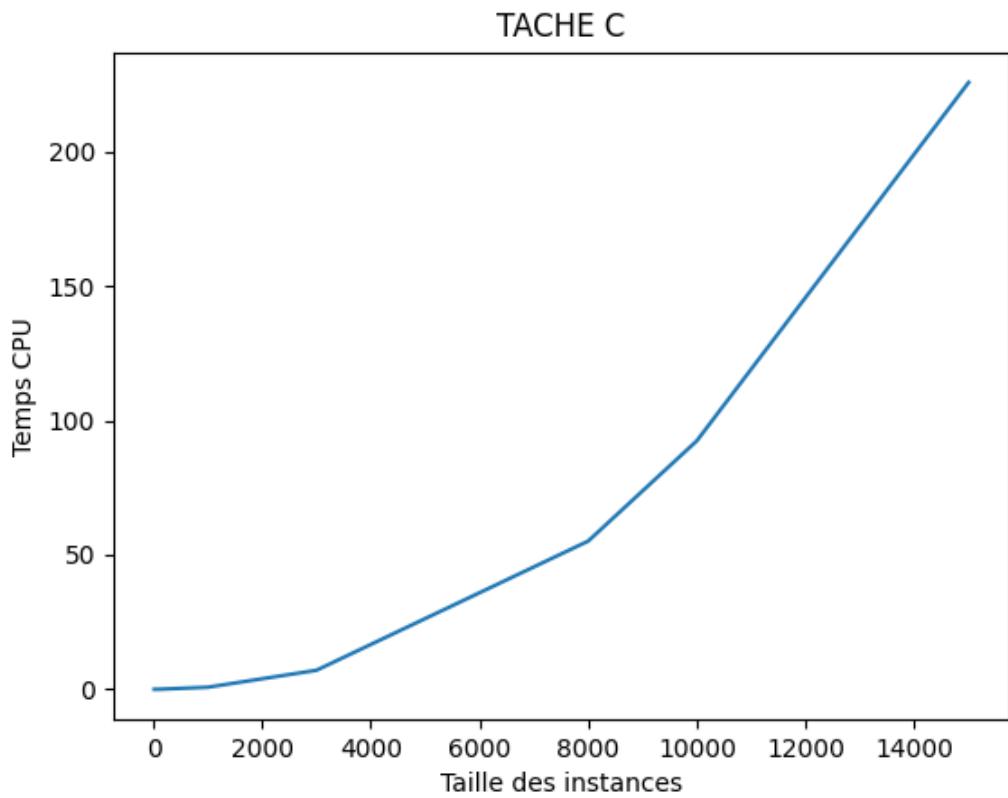
tant que cpt < n faire:
    pour j ∈ [1, ..., m] faire:
        T[i][j] = min (T[(i-1)mod(2)][j]+cdel, T[i][j-1]+cins,
                        T[(i-1)mod(2)][j-1]+csub(x[cpt], y[j-1])
    fin pour
    i = (i+1)mod(2)
    T[i][0] = T[(i - 1)mod(2)][0] + 2
    cpt = cpt + 1
fin tant que

si n pair alors:
    Retourner T[0][m]
sinon:
    Retourner T[1][m]

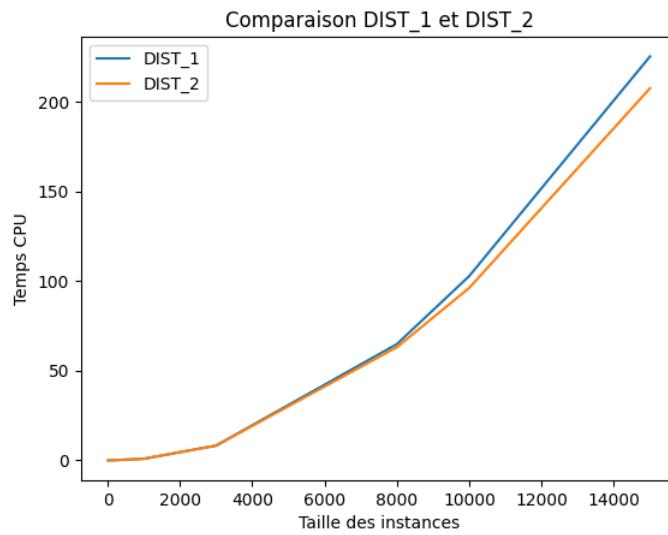
```

▼ Tâche C

Cette représentation graphique du temps d'exécution de `DIST_2` en fonction de la taille des instances est en accord avec la complexité théorique polynomiale.



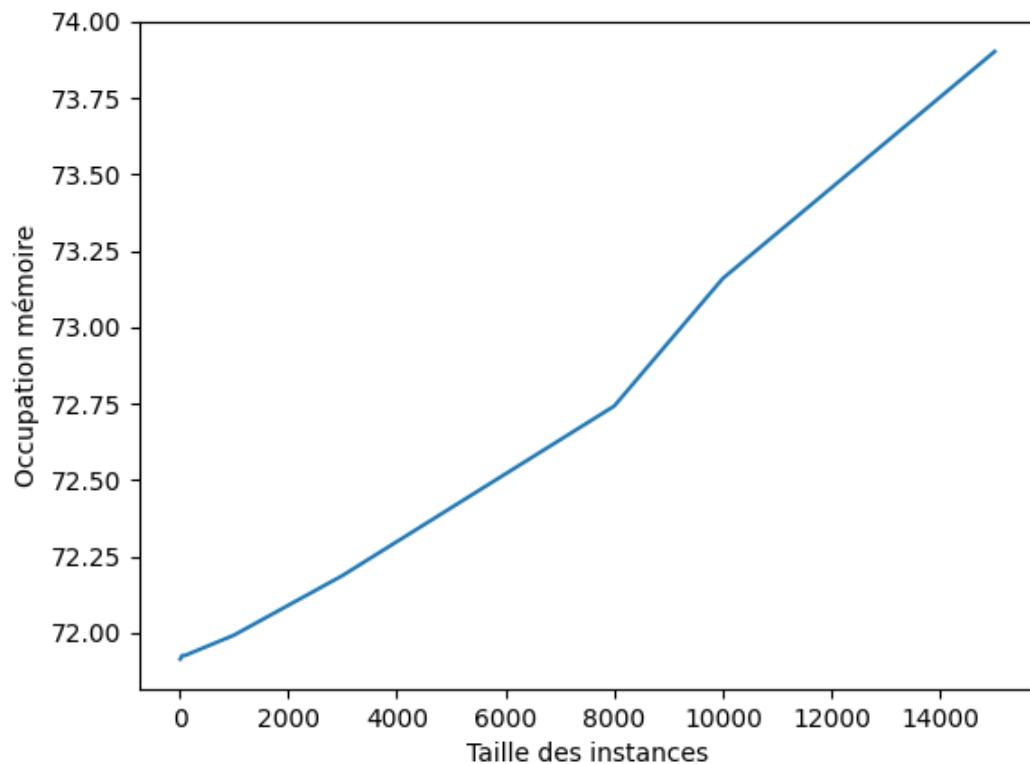
La comparaison entre les complexités temporelles de DIST_1 et DIST_2 est cohérente car le nombre de calculs est le même dans les deux fonctions.



Concernant la complexité spatiale, en reprenant la même instance que précédemment, on peut voir que la complexité spatiale théorique est nettement

plus faible et vaut : $2 * 13395 * 8 = 2,1.10^5$ bits utilisés (comparés aux $1,6.10^9$ bits de la tâche B).

`DIST_2` à donc bien une complexité spatiale linéaire ce qui est en accord avec le graphique suivant représentant l'occupation mémoire de `DIST_2` en fonction de la taille des instances:



3.4 AMÉLIORATION DE LA COMPLEXITÉ SPATIALE DU CALCUL D'UN ALIGNEMENT OPTIMAL PAR LA MÉTHODE “DIVISER POUR RÉGNER”

▼ Question 21

```
Algoritme mot_gaps:  
Entrée: un entier naturel k  
Sortie: le mot constitué de k gaps  
Retourner k*'-'
```

▼ Question 22

```
Algorithme align_lettre_mot:  
  
Entrée: x un mot de longueur l et y un mot non-vide de longueur m  
Sortie: meilleur alignement de (x,y)  
  
i = 0  
tant que i < m faire :  
    si y[i] = x alors :  
        Retourner (mot_gaps(i)+x+mot_gaps(m - i - 1) , y)  
    fin si  
    i = i + 1  
fin tant que  
  
i = 0  
tant que i < m faire :  
    si csub(x, y[i]) = 3 alors :  
        Retourner (mot_gaps(i)+x+mot_gaps(m - i - 1) , y)  
    fin si  
    i = i + 1  
fin tant que  
si i = m alors :  
    Retourner mot_gaps(i - 1)+x , y  
fin si
```

▼ Question 23

L'alignement optimal (\bar{s}, \bar{t}) de
 (x^1, y^1)

$\bar{s} : B A L$

$\bar{t} : R O -$

A un coût d'alignement de 13

L'alignement optimal (\bar{u}, \bar{v}) de
 (x^2, y^2)

$\bar{u} : L O N -$

$\bar{v} : - - N D$

A un coût d'alignement de 9

Mais l'alignement concaténé $(\bar{s}.\bar{u}, \bar{t}.\bar{v})$ de (x, y) donné par :

$\bar{s}.\bar{u} : B A L L O N -$
 $\bar{t}.\bar{v} : R O - - - N D$

N'est pas optimal.

En effet il est de coût $9 + 13 = 22$, alors que l'alignement suivant est de coût 17 :

$\bar{x} : B A L L O N -$
 $\bar{y} : - - - R O N D$

▼ Question 24

Algorithme SOL_2:

Entrée: x et y deux mots de longueurs n et m , u et v deux variables temporaires
contenant différents alignements de x et y
Sortie: (u, v) meilleur alignement calculé de (x, y)

```
i = n//2
si m = 0 et n != 0 alors :
    u = u + x
    v = v + mot_gaps(n)
sinon si n = 0 et m != 0 alors :
    u = u + mot_gaps(m)
    v = v + y
sinon si n = 1 et m >= 1 alors :
    u = u + align_lettre_mot(x,y)[0]
    v = v + align_lettre_mot(x,y)[1]
sinon si n > 1 et m >= 1 alors :
    j = coupure(x,y)
    SOL_2(x[:i],y[:j],u,v)
    SOL_2(x[i+1:],y[j+1:],u,v)
Retourner u , v
```

▼ Question 25

Algorithme Coupure:

Entrée: x et y deux mots de longueur n et m
Sortie: l'indice j^* permettant de réaliser la coupure de (x, y) sachant que
 $i^* = n$

```
coupe = n//2
cmp = 0
i1=1
i2 = 1
pour j ∈ [0, ..., m] faire :
    T[0][j] = j*cins
fin pour
T[1][0] = cdel
pour j ∈ [0, ..., m] faire :
    I[0][j] = j
    I[1][j] = j
fin pour

tant que cmp < n faire:
    pour j ∈ [1, ..., m] faire:
        T[i1][j] = min (T[(i1-1)mod(2)][j] + cdel , T[i1][j-1]+cins ,
                        T[(i1-1)mod(2)][j-1]+ csub(x[cmp],y[j-1]))

        si cmp ≥ coupe faire:
            si T[i1][j] = T[(i1-1)mod(2)][j]+cdel faire:
                I[i2][j] = I[(i2-1)mod(2)][j]
```

```

        sinon si T[i1][j] = T[i1][j-1]+cins alors:
            I[i2][j] = I[i2][j-1]
        sinon si T[i1][j] = T[(i1-1)mod(2)][j-1]+csub(x[cmp], y[j-1]) faire:
            I[i2][j] = I[(i2-1)mod(2)][j-1]
    fin pour

    si cmp ≥ coupe alors:
        i2 = (i2+1)mod(2)
        i1 = (i1+1)mod(2)
        T[i1][0] = T[(i1-1)mod(2)][0]+ 2
        cmp = cmp + 1
    fin tant que

    si (n-coupe)mod(2) = 0 alors:
        Retourner I[0][m]
    Retourner I[1][m]

```

▼ Question 26

L'exécution de l'algorithme `Coupure` correspond au remplissage de 2 matrices T et I , chacune de taille $2 \times m$. Ainsi, la complexité spatiale de `Coupure` est en $\Theta(4m) \equiv \Theta(m)$

▼ Question 27

Dans le pire cas, $n = m$ et on a $n-1$ coupures pour le couple de mots considéré. Dans ce cas, l'arbre binaire des appels récursifs est complet. Chaque sous-problème porte sur un sous-mot de taille $\leq \frac{m}{2}$, donné par l'algorithme `Coupure`.

Soit $c(m)$ la complexité de `sol_2.` $c(m)$ est majoré par $c'(m)$:

$$c'(m) = 2 c'\left(\frac{m}{2}\right) + \Theta(m)$$

Où l'opération de fusion des réponses aux sous-problèmes est la concaténation de deux sous-tableau de taille maximale $\frac{n+m}{2}$ qui prend au pire cas une taille $m \in \Theta(m)$.

D'après le Théorème maître, comme $2 = 2^1$, on a que $c'(m) \in \Theta(m^1 \log(m)) \equiv \Theta(m \log m)$. On applique l'hypothèse que $m \leq n$. On en déduit enfin que `sol_2` est en $\Theta(n \log n)$

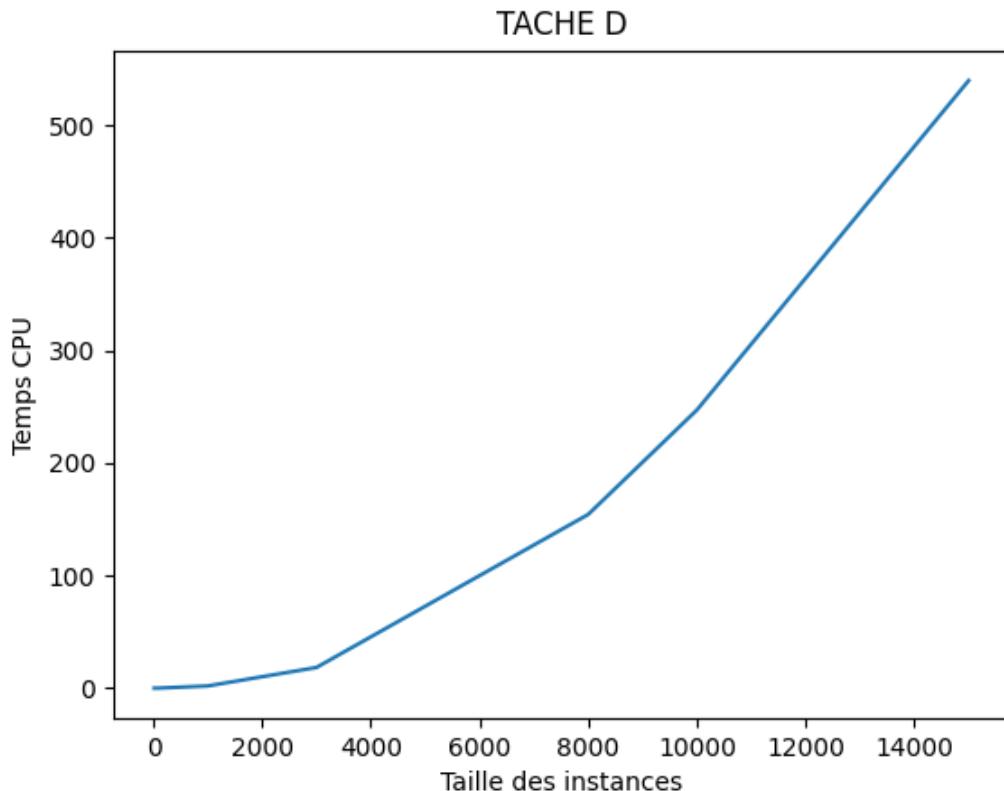
▼ Question 28

`Coupure` réalise toujours $n \times m$ calculs pour calculer les valeurs des cases du tableau T qui donne la distance d'édition ainsi que du tableau I les coupures du couple de mot étudié. On peut considérer que les calculs et mises à jour, se font en temps constant (et ainsi négliger ce qu'il se passe dans I à partir de $i^* = \frac{n}{2}$)

L'algorithme `Coupure` est donc en complexité temporelle $\Theta(nm)$.

▼ Tâche D

Cette représentation graphique du temps d'exécution de `SOL_2` en fonction de la taille des instances montre une évolution polynomiale ce que qui est cohérent avec la complexité théorique.



On peut estimer la consommation mémoire de `SOL_2` en considérant que `coupure` libère son allocation en fin d'appel. De ce fait, comme les résultats sont stockés dans deux listes de taille au plus $n + m$, la quantité de bits utilisé est $2 * (n + m) * 8$. Pour l'instance `Inst_0015000.adn`, la consommation mémoire est donc $2 * (15000 + 13395) * 8 = 401850$ bits.

▼ Question 29

L'étude des complexités temporelles des tâches B et D nous montre que l'amélioration de la complexité spatiale proposée à la tâche D se fait au détriment de la complexité temporelle. En effet, pour la tâche D, on remarque clairement sur le graphe comparatif ci-dessous que la complexité temporelle de D est supérieure à celle de B (la courbe correspondante croît plus vite), quand bien même les algorithmes appartiennent aux mêmes classes de complexité.

