LU3IN003 : Rapport de Projet

L'Alignement de séquences ADN

```
AAB24882        TYHMCQFHCRYVNNHSGEKLYECNERSKAFSCPSHLQCHKRRQIGEKTHEHNQCGKAFPT 60
AAB24881        --------------------YECNQCGKAFAQHSSLKCHYRTHIGEKPYECNQCGKAFSK 40
                                    ****: .***:  * *:** * :****.:* *******..

AAB24882        PSHLQYHERTHTGEKPYECHQCGQAFKKCSLLQRHKRTHTGEKPYE-CNQCGKAFAQ- 116
AAB24881        HSHLQCHKRTHTGEKPYECNQCGKAFSQHGLLQRHKRTHTGEKPYMNVINMVKPLHNS 98
                 **** *:***********:***:**.: .***************    :  *.: :
```

Adchayan THAMBIAIAH & Rahmine Nedir

Rapport de Projet : Alignement de séquences LU3IN003 - Sorbonne Université Automne 2022

Ce projet se porte sur l'alignement de séquences ADN. Cela consiste à mesurer les similarités entre deux séquences d'ADN. Les séquences sont représentées par l'alphabet {A,T,G,C} et sont vues comme des mots. On se ramène alors sur deux problèmes algorithmiques de texte qui sont le calcul de distance d'édition entre deux mots et la réalisation d'un alignement de cette distance. Le projet sera dans l'ensemble en langage de programmation Python.

2 Le problème d'alignement de séquences

2.2 Alignement de deux mots

Définitions 2.1 :

Un alphabet est un ensemble fini de caractères. Soit Σ un alphabet non vide. Un mot sur Σ est une suite finie de caractères de Σ. On notera $\Sigma*$ l'ensemble des mots sur Σ. Pour $x \in \Sigma*$, la longueur du mot x, notée $|x|$, est le nombre de caractères dans x comptés avec multiplicité. Si $|x|=n$, les caractères de x seront indexés par [1..n], le mot x s'écrit alors x1 x2 . . . xn. Soient x et y deux mots sur Σ de longueurs respectives m et n. La concaténation des mots x et y, notée x·y, est le mot x1 x2 . . . xm y1 y2 . . . yn. Quel que soit l'alphabet, il existe un seul mot de longueur nulle appelé le mot vide et noté ε.

Définition 2.2

Soit $(x,y) \in \Sigma^* \times \Sigma^*$. *Soit* $(\bar{x},\bar{y}) \in \overline{\Sigma}^* \times \overline{\Sigma}^*$.
On dit que $(\bar{x},\bar{y})$ *est un* ***alignement*** *de* (x,y) *ssi*

$$\begin{cases} (i)\ \ \pi(\bar{x}) = x \\ (ii)\ \ \pi(\bar{y}) = y \\ (iii)\ \ |\bar{x}| = |\bar{y}| \\ (iv)\ \ \forall i \in [1..|\bar{x}|],\ \bar{x}_i \neq - \text{ ou } \bar{y}_i \neq - \end{cases}$$

On appellera longueur de l'alignement $(\bar{x},\bar{y})$, *la longueur de* $\bar{x}$.

Exemple : Pour $\Sigma = \{A,T,G,C\}$, x=ATTGTA et y=ATCTTA, voici plusieurs exemples d'alignements de (x,y) :

$\bar{x}$: A T T G T A	$\bar{x}$: A T − T G T A	$\bar{x}$: A T T G T − A	$\bar{x}$: − − − − − A T T G T A
$\bar{y}$: A T C T T A	$\bar{y}$: A T C T − T A	$\bar{y}$: A T − C T T A	$\bar{y}$: A T C T T A − − − − −

Question 1 :

Prenons x = ATTGTA, y = ATCTTA, u = ACTAT, v = TGCAT.

$\overline{x}$ = ATTGT_A

$\overline{y}$ = AT_CTTA

$(\overline{x},\overline{y})$ est bien un alignement de (x,y) car $\pi(\overline{x}) = x$,$\pi(\overline{y}) = y$ et $|\overline{x}| = |\overline{y}| = n$ d'après la définition 2.2 (exemple précèdent).

$\overline{u}$ = ACT_AT

$\overline{v}$ = TG_CAT

$(\overline{u}, \overline{v})$est bien un alignement de (u, v) d'après la définition de 2.2 puisqu'en retirant les gaps dans $(\overline{u}, \overline{v})$, on revient bien sur (u, v) et $|\overline{u}| = |\overline{v}| = m$.

En effectuant la concaténation sur les mots entre $\overline{x}$ et $\overline{u}$ puis $\overline{y}$ et $\overline{v}$ soit donc $\overline{x}.\overline{u}$ et $\overline{y}.\overline{v}$, on obtient d'après la définition 2.1 :

$\overline{x}.\overline{u}$ = ATTGT_AACT_AT

$\overline{y}.\overline{v}$ = AT_CTTATG_CAT

En retirant les gaps on a :

$\pi(\overline{x}.\overline{u})$ = ATTGTAACTAT = $x.u$

$\pi(\overline{y}.\overline{v})$ = ATCTTATGCAT = $y.v$

avec $|\overline{x}.\overline{u}| = |\overline{y}.\overline{v}| = n + m$

On peut conclure que si $(\overline{x}, \overline{y})$ et $(\overline{u}, \overline{v})$ sont respectivement des alignements de (x, y) et (u, v) alors $(\overline{x}.\overline{u}, \overline{y}.\overline{v})$ est bien un alignement de $(x.u, y.v)$.

Question 2 :

Si $x \in \Sigma^*$est un mot de longueur n et $y \in \Sigma^*$est un mot de longueur m. Le majorant de longueur maximum obtenu pour un alignement de (x, y) est de longueur n+m car dans les pires cas on a donc m gaps devant le mot x et n gaps après le mot y . Prenons en exemple $\overline{x}$ = ATT_CGT et $\overline{y}$ = ATCTA_GTA avec $|\overline{x}|$= 7 et $|\overline{y}|$ = 9. Un alignement de (x, y) est donc :

$\overline{x}$ = ATT_CGT_ _ _ _ _ _ _ _ _

$\overline{y}$ = _ _ _ _ _ _ _ATCTA_GTA

On a alors $(\overline{x}, \overline{y})$ un alignement de (x, y) de longueur $|\overline{x}| = |\overline{y}| = 16$

2.3 Coût d'un alignement et distance d'édition

Un alignement $(\bar{x}, \bar{y})$ de (x, y) peut aussi être vu comme décrivant une manière de transformer le mot x en y en utilisant les trois opérations suivantes (appelées opérations d'édition) :

- l'**insertion**, qui consiste à insérer une lettre de y dans le mot x, est encodée par un gap dans $\bar{x}$;
- la **suppression**, qui consiste à supprimer une lettre du mot x, est encodée par un gap dans $\bar{y}$;
- la **substitution**, qui consiste à changer une lettre de x en une lettre de y, est encodée par deux lettres différentes à la même position dans $\bar{x}$ et $\bar{y}$.

On fixe donc $c_{ins} \in \mathbb{R}^+$ (resp. $c_{del} \in \mathbb{R}^+$) le coût d'une insertion (resp. d'une suppression), et pour tout couple de lettres différentes $(a,b) \in \Sigma^2$, $c_{sub}(a,b) \in \mathbb{R}^+$ le coût de la substitution de b à a (on remplace a par b). Pour des raisons pratiques, on s'autorise à écrire $c_{sub}(a,b)$ même si $a = b$, c'est-à-dire même si l'opération de substitution de a par b revient à ne rien faire, et ne coûte donc rien. On pose donc $\forall a \in \Sigma,\ c_{sub}(a,a) = 0$.

Définition 2.3

Soit $(\bar{x}, \bar{y})$ un alignement de $(x,y) \in \Sigma^* \times \Sigma^*$ de longueur l.
*Le **coût** de l'alignement $(\bar{x}, \bar{y})$, noté $C(\bar{x}, \bar{y})$ est défini comme suit :*

$$C(\bar{x},\bar{y}) = \sum_{k=1}^{l} c(\bar{x}_k, \bar{y}_k) \quad \text{où} \quad \forall (a,b) \in \overline{\Sigma}^2 \setminus \{(-,-)\},\ c(a,b) = \begin{cases} c_{ins} & \text{si } a = - \\ c_{del} & \text{si } b = - \\ c_{sub}(a,b) & \text{sinon} \end{cases}$$

Définition 2.4

Soit $(x,y) \in \Sigma^* \times \Sigma^*$.
*La **distance d'édition de x à y**, est $d(x,y) = \min\left\{C(\bar{x},\bar{y}) \mid (\bar{x},\bar{y}) \text{ est un alignement de } (x,y)\right\}$.*

Remarque : Notez que telle que définie, d n'est pas une distance au sens mathématique du terme, elle n'est même pas symétrique a priori. En revanche si $c_{del} = c_{ins}$, et si c_{sub} est une distance sur Σ, alors d est bien une distance.

Exemple : On considère l'alphabet $\Sigma = \{A,C,T,G\}$ avec les paramètres de coût suivants : $c_{del} = c_{ins} = 2$, $c_{sub}(a,b) = 3$ si $\{a,b\}$ est une paire concordante, c'est-à-dire $\{a,b\} = \{A,T\}$ ou $\{a,b\} = \{G,C\}$ et $c_{sub}(a,b) = 4$ sinon.

Pour x =ATTGTA et y =ATCTTA, $d(x,y) = 4$, et cette distance d'édition est atteinte notamment par l'alignement minimal suivant :

$\bar{x}$:	A	T	−	T	G	T	A
$\bar{y}$:	A	T	C	T	−	T	A

3. Algorithmes pour l'alignement de séquences

Sous le nom "alignement de séquences" se cachent en réalité deux problèmes : le calcul de la distance d'édition et la production d'un alignement de coût minimal.

DIST — Entrée : $\boldsymbol{x}$ et $\boldsymbol{y}$ deux mots sur Σ ; Sortie : $d(x,y)$

ALI — Entrée : $\boldsymbol{x}$ et $\boldsymbol{y}$ deux mots sur Σ ; Sortie : $(\bar{x},\bar{y})$ un alignement de (x,y) tel que $C(\bar{x},\bar{y}) = d(x,y)$

3.1 Méthode naïve par énumération

Question 3 :

Pour $x \in \Sigma^*$ un mot de longueur n. En ajoutant à x exactement k gaps, on obtient plusieurs combinaisons de mots $\overline{x}$ possible soit :

$$\frac{(n+k)\times((n+k)-1)\ldots\times((n+k)-k+1)}{k\times(k-1)\ldots\ldots\times 1} = \frac{(n+k)!}{k!(n+k-k)!} = \binom{n+k}{k} = C_k^{n+k}$$

On a donc une combinaison de $\binom{n+k}{k}$ de mot possible pour $\overline{x}$ avec exactement k gaps (on obtient un coefficient binomial).

Question 4 :

Pour un couple de mots (x, y) de longueurs n et m . Après l'ajout de k gaps à x pour obtenir un mot de $\overline{x} \in \overline{\Sigma^*}$.On ajoute $n + k - m$ gaps à y pour l'obtention d'un mot de $\overline{y} \in \overline{\Sigma^*}$ sans chevaucher les gaps de $\overline{x}$. Le nombre de mots possible pour $\overline{y}$ se réduit à (coefficient binomial) :

$\frac{(n+k)!}{k!n!} \times \frac{n!}{k'!(n-k')!} = C_k^{n+k} \times C_{k'}^n$ avec $k' = n + k - m$

En connaissant le nombre de mots possible pour $\overline{y}$, le nombre d'alignements possibles de (x, y) est donc :

$\sum_{k=0}^{m} \left(C_k^{n+k} \times C_{k'}^n\right)$ avec $k' = n + k - m$

Le nombre d'alignement pour $|x| = 15$ et $|y| = 10$ est

Question 5 :

Le temps nécessaire pour un algorithme naïf énumérant tous les alignements de deux mots pour trouver la distance d'édition serait en $\tau \times \sum_{k=0}^{m} \left(C_k^{n+k} \times C_{n+k-m}^n\right)$ soit une complexité en factorielle.

Pour chaque alignement, on calcule le coût minimal d'un alignement. Le temps nécessaire pour le coût est noté c .Donc le temps nécessaire d'un algorithme prenant en compte le calcul du coût minimal est de $(\tau \times c) \sum_{k=0}^{m} \left(C_k^{n+k} \times C_{n+k-m}^n\right)$ soit une complexité en factorielle.

Question 6 :

La place mémoire requise pour un algorithme naïf qui consisterait à énumérer tous les alignements de deux mots pour trouver la distance d'édition serait au pire cas $\gamma \times x$ nombre d'alignements en place mémoire avec $\gamma = |x| + |y|$ la cumulation des longueurs de x et y . Une complexité en factorielle.

Pour un alignement minimal en mémoire il ne faudra pas d'espace mémoire ne plus. La complexité reste inchangée.

Lorsque cette fonction est appelée pour $i = |x|$ et $j = |y|$ avec $c < dist$, elle met à jour la valeur $dist$ à c, qui représente alors le coût d'un alignement de (x, y).
On définit également une fonction `DIST_NAIF` qui à partir d'un couple (x, y) appelle simplement `DIST_NAIF_REC`$(x, y, 0, 0, 0, dist)$ pour une variable $dist$ initialisée à $+\infty$ (min de l'ensemble vide).

`DIST_NAIF`

Entrée : x et y deux mots
Sortie : $d(X, Y)$
retourner `DIST_NAIF_REC`$(x, y, 0, 0, 0, +\infty)$

`DIST_NAIF_REC`

Entrée : x et y deux mots,
i un indice dans $[0..|x|]$, j un indice dans $[0..|y|]$,
c le coût de l'alignement de $(x_{[1..i]}, y_{[1..j]})$
$dist$ le coût du meilleur alignement de (x, y) connu avant cet appel
Sortie : $dist$ le coût du meilleur alignement de (x, y) connu après cet appel

`Si` $i = |x|$ et $j = |y|$
`Alors Si` $(c < dist)$ `alors` $dist \leftarrow c$
`Sinon`
  `Si` $(i < |x|)$ et $(j < |y|)$
  `Alors` $dist \leftarrow$ `DIST_NAIF_REC`$(x, y, i+1, j+1, c + c_{sub}(x_{i+1}, y_{j+1}), dist)$
  `Si` $(i < |x|)$
  `Alors` $dist \leftarrow$ `DIST_NAIF_REC`$(x, y, i+1, j, c + c_{del}, dist)$
  `Si` $(j < |y|)$
  `Alors` $dist \leftarrow$ `DIST_NAIF_REC`$(x, y, i, j+1, c + c_{ins}, dist)$
retourner $dist$

Remarque importante : Dans toutes les expérimentations numériques du projet, l'alphabet utilisé sera $\Sigma = \{A,C,T,G\}$ avec les paramètres de coût suivants : $c_{del} = c_{ins} = 2$, $c_{sub}(a, b) = 0$ si $a = b$, $c_{sub}(a, b) = 3$ si $\{a, b\}$ est une paire concordante, c'est-à-dire $\{a, b\} = \{A,T\}$ ou $\{a, b\} = \{G,C\}$, et $c_{sub}(a, b) = 4$ sinon.

Tache A :

• Certaines fonctions ne seront pas présente sur le rapport pour éviter de le surcharger car très peu utile mais ils seront consultables dans les fichiers codes appropriés du projet).

distance.py

```python
#structure contenant les séquences x et y
def Struct_Instance(lignes):


    Sequences = {

        "X" : (str.strip(lignes[2])).split(' '),
        "Y" : (str.strip(lignes[3])).split(' '),
    }

    return Sequences

#(liste[char] X,liste[char] Y, int i, int j, int c, int dist) -> dist : Calcule
recursivement la distance des mots x et y naivement
def DIST_NAIF_REC(X,Y,i,j,c,dist):
```

```
    if (i == (len(X)-1)) and (j == (len(Y)-1)):
        if(c<dist): dist = c

    else:

        if((i<(len(X)-1)) and (j<(len(Y)-1))):
            dist = DIST_NAIF_REC(X,Y,i+1,j+1,c+csub(X[i],Y[j]),dist)
        if (i<(len(X)-1)):
            dist = DIST_NAIF_REC(X,Y,i+1,j,c+cdel(),dist)
        if (j<(len(Y)-1)):
            dist = DIST_NAIF_REC(X,Y,i,j+1,c+cins(),dist)

    return dist

# void -> int : cout d'une suppression
def cdel():
    return 2
# void -> int : cout d'une insertion
def cins():
    return 2
# void -> int : cout d'une substitution
def csub(a,b):

    if(a==b):
        return 0
    if(( a=='A' and b=='T' ) or ( a =='G' and b =='C' ) or (a=='T' and b=='A')
or (a == 'C' and a == 'G')):
        return 3
    else:
        return 4
```

La structure de donnée implémenter pour les séquences d'ADN sont les dictionnaires car elle permette une bonne manipulation des informations ici. Elle contient les séquences x et y. Elle est assez similaire à une structure présente en C d'où son utilisation. Voici ci-dessus les pseudo-codes retranscrit sous Python.

test.py

• Le fichier test.py regroupe le code permettant de tester la validité de l'implémentation de *DIST_NAiF_REC* avec les instances ***Inst_0000010_44.adn***, ***Inst_0000010_7.adn*** et ***Inst_0000010_8.adn***.

Voici le résultat obtenu dans le terminal après l'exécution du fichier :

```
PS C:\Users\adcha\Desktop\Projet Algo> python .\test.py
Sequences_Inst_0000010_44:
X: ['T', 'A', 'T', 'A', 'T', 'G', 'A', 'G', 'T', 'C']
Y: ['T', 'A', 'T', 'T', 'T']
taille_X: 10
taille_Y: 5
distance:10

Sequences_Inst_0000010_7:
X: ['T', 'G', 'G', 'G', 'T', 'G', 'C', 'T', 'A', 'T']
Y: ['G', 'G', 'G', 'G', 'T', 'T', 'C', 'T', 'A', 'T']
taille_X: 10
taille_Y: 10
distance:8

Sequences_Inst_0000010_8:
X: ['A', 'A', 'C', 'T', 'G', 'T', 'C', 'T', 'T', 'T']
Y: ['A', 'A', 'C', 'T', 'G', 'T', 'T', 'T', 'T']
taille_X: 10
taille_Y: 9
distance:2

PS C:\Users\adcha\Desktop\Projet Algo>
```

On constate que l'implémentation est bien valide sur ces instances car les distances d'éditions retourner sont bien 10, 8 et 2. (les fichiers calcul_distance1 et calcul_distance2 permette de vérifier une instance depuis le lancement du ficher avec un fichier instance en argument ou par le dossier instance_genome_test contenant toutes les instances à vérifier et puis par lancement de calcul_distance2 qui va récupère les fichiers instance dans ce dossier)

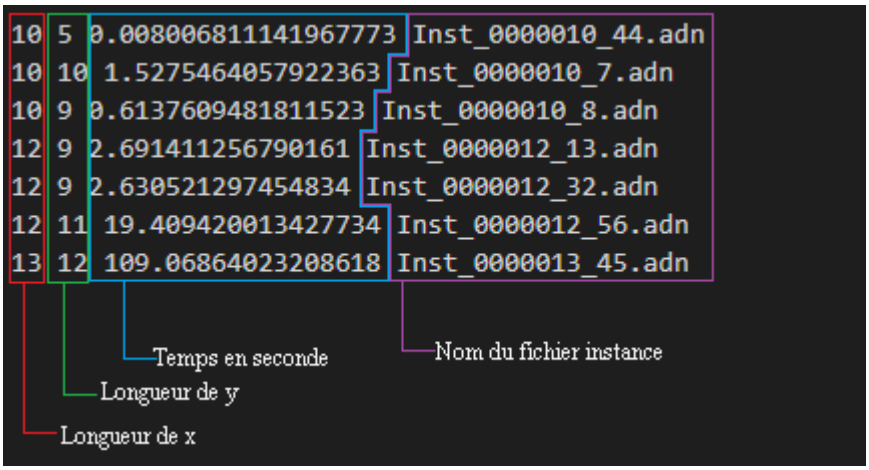
time_test.py

• Ce code permet d'évaluer la performance de calcule des différentes distances d'éditions des instances de génome contenue dans le fichier *Instances_genome* au format **.adn** et l'exécution du code s'arrête lorsque la résolution pour les instances fournies dépasse une minute. Les temps de calcul sont écrits dans le fichier *exectime_DIST_NAIF.txt.*

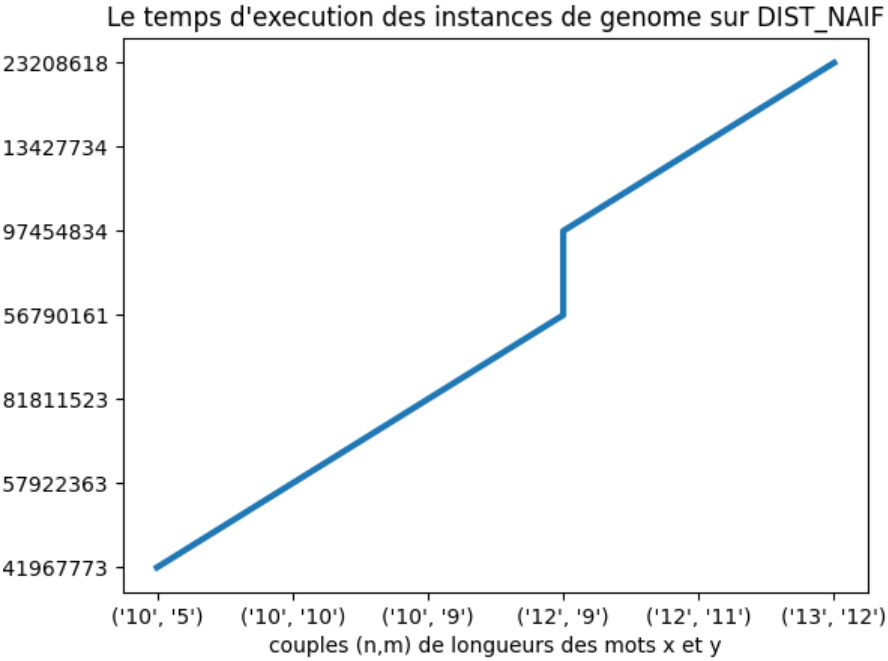
```
import time
import distance
import os


def main():


    folderpath = "./Instances_genome/" # dossier contenant les fichiers de
couple de sequence de genomes a resoudre
    fichiers = os.listdir(folderpath)
    timefile = open('exectime_DIST_NAIF.txt','w') #ouverture du fichier qui
va contenir tous les données de temps resolution. #format fichier ->
@taille_x @taille_y @temps @nom_du_fichier
    i = 0
    start_time = 0
    end_time = 0

    #on lit chaque fichier et on calcule le temps nécessaire pour calculer
la distance naivement par DIST_NAIF
    while(i<len(fichiers) and (end_time-start_time)<60):

        if(os.path.isfile(os.path.join(folderpath,fichiers[i])) and
os.path.splitext(fichiers[i])[1] == ".adn"):

            fichierO = open(os.path.join(folderpath,fichiers[i]),'r')
            lignes = fichierO.readlines()
            struct_sequences = distance.Struct_Instance(lignes)
            print(fichiers[i])

            start_time = time.time()
            dist =
distance.DIST_NAIF(struct_sequences["X"],struct_sequences["Y"])
            end_time = time.time()

            fichierO.close()
            print(str(end_time-start_time))
            timefile.write(str(len(struct_sequences["X"]))+'
'+str(len(struct_sequences["Y"]))+' '+str((end_time-start_time))+'
'+str(fichiers[i])+"\n")

        i+=1

    timefile.close()

    return

main()
```

exectime_DIST_NAIF.txt

Un exemple de fichier en sortie



On obtient le graphe suivant :



En faisant un Plot du fichier ***exectime_DIST_NAIF.txt*** avec le code situé dans le fichier ***graph_plotting.py.*** On constate que le temps d'exécution grimpe assez rapidement et ceci est valide puisque la complexité temporelle est de type factorielle. (Le fichier graph_plotting.py prend en entrée le nom du fichier, un label un label y et un titre).

• L'estimation de la consommation mémoire au fonctionnement de cette méthode est d'environ 48.980 MB pour l'instance des génomes du fichier **Inst_0000010_44.adn.** On est passé par l'utilisation du module python memory_profiler pour visualiser la consommation mémoire de l'algorithme pour cette instance. Voici une capture écran de ce résultat détaillé.

```
PS C:\Users\adcha\Desktop\Projet Algo> python -m memory_profiler .\calcul_distance1.py ./Instances_genome/Inst_0000010_44.adn
X: ['T', 'A', 'T', 'A', 'T', 'G', 'A', 'G', 'T', 'C']
Y: ['T', 'A', 'T', 'T', 'T']
taille_X: 10
taille_Y: 5
distance:10
Filename: .\distance.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    15   48.980 MiB   48.980 MiB           1   @profile
    16                                         def DIST_NAIF(X,Y):
    17   48.828 MiB   48.828 MiB           1       return DIST_NAIF_REC(X,Y,0,0,0,100000000)


Filename: .\distance.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    20   48.980 MiB -2048.488 MiB       18182   @profile
    21                                         def DIST_NAIF_REC(X,Y,i,j,c,dist):
    22
    23   48.980 MiB -2097.469 MiB       18182       if (i == (len(X)-1)) and (j == (len(Y)-1)):
    24   48.980 MiB -654.012 MiB        5641            if(c<dist): dist = c
    25
    26                                             else:
    27
    28   48.980 MiB -1443.457 MiB       12541           if((i<(len(X)-1)) and (j<(len(Y)-1))):
    29   48.980 MiB -326.473 MiB        2820                dist = DIST_NAIF_REC(X,Y,i+1,j+1,c+csub(X[i],Y[j]),dist)
    30   48.980 MiB -1443.609 MiB       12541           if (i<(len(X)-1)):
    31   48.980 MiB -1280.906 MiB       11181               dist = DIST_NAIF_REC(X,Y,i+1,j,c+cdel(),dist)
    32   48.980 MiB -1444.523 MiB       12541           if (j<(len(Y)-1)):
    33   48.980 MiB -491.309 MiB        4180                dist = DIST_NAIF_REC(X,Y,i,j+1,c+cins(),dist)
    34
    35   48.980 MiB -2098.840 MiB       18182       return dist
```

3.2 Programmation dynamique

3.2.1 Calcul de la distance d'édition par programmation dynamique

Pour les questions qui suivent, on considère $(x, y) \in \Sigma^* \times \Sigma^*$ un couple de mots de longueurs respectives n et m. Pour $i \in [0..n]$ et $j \in [0..m]$, on introduit les deux notations suivantes :

$$D(i,j) = d(x_{[1..i]}, y_{[1..j]}) \quad \text{et} \quad Al(i,j) = \{(\bar{u}, \bar{v}) \mid (\bar{u}, \bar{v}) \text{ est un alignement de } (x_{[1..i]}, y_{[1..j]})\}$$

On a donc $D(n,m) = d(x,y)$.

Question 7 :

Soit $(\overline{u}, \overline{v})$ un alignement de $(x_{[1\ldots i]}, y_{[1\ldots j]})$ de longueur l avec $l < i$ et $l < j$.

- Si $\overline{u_l} = -$ alors $\overline{v_l} = y_l$
- Si $\overline{v_l} = -$ alors$\overline{u_l} = x_l$
- Si $\overline{u_l} \neq -$ et $\overline{v_l} \neq -$ alors $\overline{u_l} = x_l$ et $\overline{v_l} = y_l$

Question 8 :

En distinguant les trois cas de la question 7, on peut exprimer $C(\overline{u}, \overline{v})$ à partir de $C(\overline{u}_{[1\ldots l-1]}, \overline{v}_{[1\ldots l-1]})$ par :

- $C(\overline{u}_{[1\ldots l-1]}, \overline{v}_{[1\ldots l-1]}) + C_{ins}$ si $\overline{u_l} = -$

$C(\overline{u},\overline{v}) =$ - $C\left(\overline{u}_{[1\ldots l-1]},\overline{v}_{[1\ldots l-1]}\right) + C_{del}$ si $\overline{v}_l = -$

- $C\left(\overline{u}_{[1\ldots l-1]},\overline{v}_{[1\ldots l-1]}\right) + C_{sub}(\overline{u}_l,\overline{v}_l)$ sinon si $\overline{u}_l \neq -$ et $\overline{v}_l \neq -$

Question 9 :

Pour $i \in [1 \ldots n]$ et $j \in [1 \ldots m]$ on peut déduire d'après les questions 7 et 8 précédentes l'expression de $D(i,j)$ avec $D(i',j')$ où $i' \leq i, j' \leq j$ et $(i',j') \neq (i,j)$. On a alors :

- Si $j' > 0$ alors $D(i',j') = D(i',j'-1) + C_{ins}$

- Si $i' > 0$ alors $D(i',j') = D(i-1,j) + C_{del}$

- Si $j' > 0$ et $i' > 0$ alors $D(i',j') = D(i'-1,j'-1) + C_{sub}\left(x_{i'}y_{j'}\right)$

Question 10 :

$D(0,0) = 0$ Puisque pour aller d'un mot vide ϵ a un autre mot vide ϵ on réalise aucune opération.

Question 11 :

D'après la question précédente 9, Pour $j \in [1 \ldots m]$, $D(0,j) = j.C_{ins}$ puisqu'on réalise une insertion pour l'opération de passage de $D(i,j-1)$ à $D(i,j)$.

De même, Pour $i \in [1 \ldots m]$, $D(i,0) = i.C_{del}$ puisqu'on réalise une suppression pour l'opération de passage de $D(i-1,j)$ à $D(i,j)$.

Question 12 :

D'après la question 9,10 et 11, le pseudo-code obtenu pour l'algorithme itératif DIST_1 est :

Dist_1(x ,y):

- *Déclaration et allocation d'une matrice D de taille* $(n+1) \times (m+1)$
- *Initialisation :*
  - $D(0,0) = 0$
  - $D(0,j) = j.C_{ins}$, *pour* $\forall\, j \in [1 \ldots m]$
  - $D(i,0) = i.C_{del}$, *pour* $\forall\, i \in [1 \ldots n]$
- *Parcours de toutes les cases de D, pour chaque case :*

  $Min\left(D_{[i][j-1]} + C_{ins}, D_{[i-1][j]} + C_{del}, D_{[i-1][j-1]} + C_{sub}(x_i,y_j)\right)$
- *On retourne à la fin la distance d'édition de* (x,y) *et la matrice* D *:*

  D *et* $D_{[n][m]}$ *retourner*

Question 13 :

Dans DIST_1, on prend en compte uniquement une matrice de taille $n \times m$ donc la complexité spatiale de DIST_1 est en $\theta(n \times m)$.

Question 14 :

L'algorithme DIST_1 est un algorithme itératif composé de deux boucle imbriqué constitué d'opérations élémentaire. L'algorithme DIST_1 effectue $n \times m$ itération. Donc DIST_1 est de complexité temporelle de $\theta(n \times m)$.

3.2.2 Calcul d'un alignement optimal par programmation dynamique

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

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

Question 15 :

Soit $(i,j) \in [0 \dots n][0 \dots . m]$, montrons que :

- si $j > 0$ et $D(i,j) = D(i,j-1) + C_{ins} \Longrightarrow \forall(\overline{s},\overline{t}) \in Al^*(i,j-1), (\overline{s}.-, \overline{t}.y_j) \in Al^*(i,j)$

On sait que si $j > 0$ alors $D(i,j) = D(i,j-1) + C_{ins}$ et $D(i,j-1)$ représente des alignements optimaux $(\overline{u},\overline{v})$ de $(x_{[1\dots i]}y_{[1\dots j-1]})$. Donc $(\overline{u},\overline{v}) \in Al^*(i,j-1)$.

$D(i,j) = D(i,j-1) + C_{ins}$ représente le passage de $y_{[1\dots.j-1]}$ vers $y_{[1\dots.j]}$ par l'insertion dans $x_{[1\dots.i]}$ d'un gap et par l'insertion dans $y_{[1\dots.j-1]}$ de y_j.

Donc $\forall(\overline{u},\overline{v}) \in Al^*$, $(\overline{u}.-, \overline{v}.y_j)$ est un alignement optimal de $(x_{[1\dots i]}, y_{[1\dots j]})$ alors $(\overline{u}.-, \overline{v}.y_j) \in Al^*(i,j)$.

Question 16 :

$SOL1(x,y,D)$:

- Soit $(x,y) \in \Sigma^* \times \Sigma^*$, $(\overline{x},\overline{y}) \in \overline{\Sigma^*} \times \overline{\Sigma^*}$ et une matrice distances d'éditions D de taille $(n+1) \times (m+1)$.

On prend $i = n$ et $j = m$

Si $i = 0$ on retourne $\overline{x} = j.-$ et $\overline{y} = \overline{y}$
Si $j = 0$ on retourne $\overline{x} = \overline{x}$ et $\overline{y} = i \cdot -$

Tant que $i > 0$ Et $j > 0$ faire
  Choisir le $Min(D[i-1][j], D[i][j-1], D[i-1][j-1])$
  Si $D[i-1][j]$ alors $\overline{x} = \overline{x}.x_i$ et $\overline{y} = \overline{y}.-$ puis $i = i+1$
  Si $D[i][j-1]$ alors $\overline{x} = \overline{x}.-$ et $\overline{y} = \overline{y}.y_j$ puis $j = j+1$
  Si $D[i-1][j-1]$ alors $\overline{x} = \overline{x}.x_i$ et $\overline{y} = \overline{y}.y_j$ puis $i = i+1$ et $j = j+1$
On inverse et on retourne $\overline{x}$ et $\overline{y}$

Question 17 :

La complexité temporelle de **DIST_1** est en $\theta(n \times m)$. Au pire cas, **SOL_1** réalise $(n+m)$ itérations et exécute à chaque itération des opérations élémentaires.

En combinant les deux algorithmes, on résout le problème **ALI** avec une complexité temporelle en $\theta(n \times m)$.

Question 18 :

D'après les questions précédentes, la complexité spatiale de **DIST_1** est en $\theta(n \times m)$. La matrice D créer par **DIST_1** est réutilisé dans **SOL_1** donc la complexité spatiale de SOL_1 est en $\theta(1)$. Donc la complexité spatiale pour résoudre **ALI** est en $\theta(n \times m)$.

Tache B :

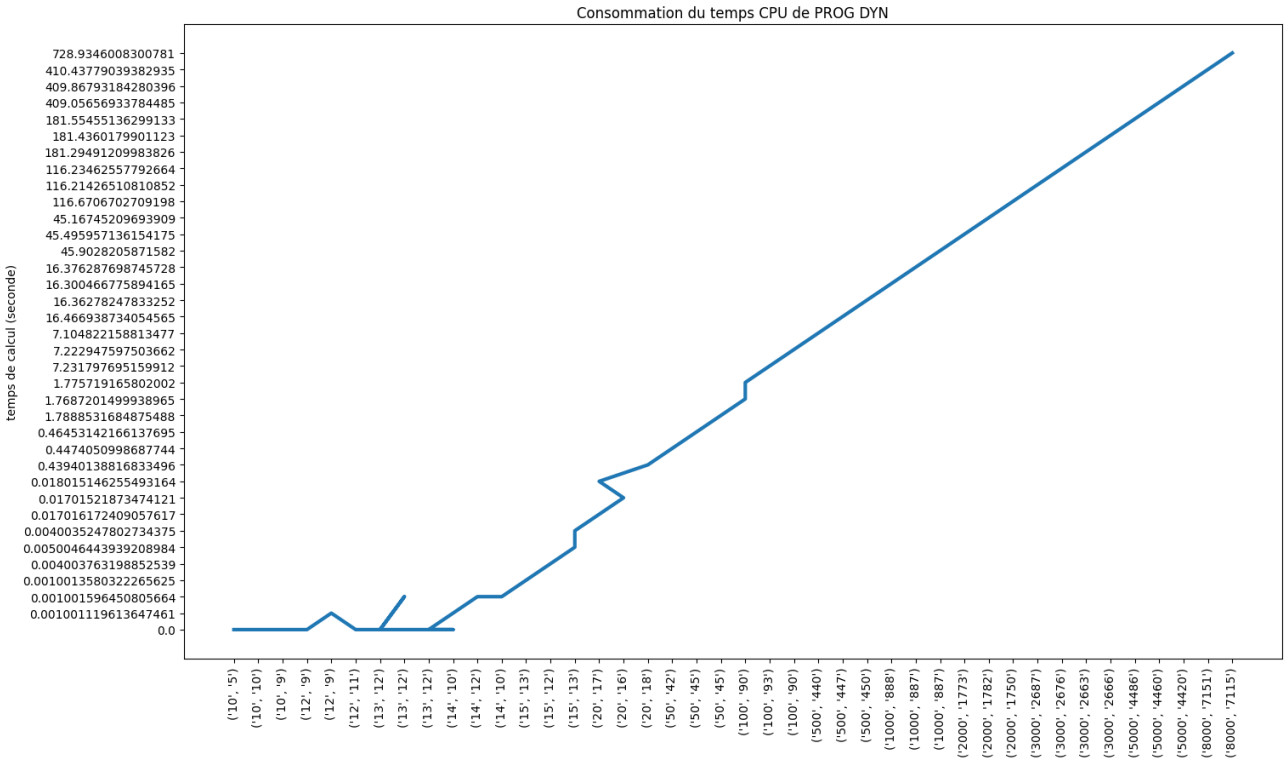
Les fonctions DIST_1, SOL_1 et PROG_DYN peuvent être retrouver dans les fichiers au même nom que les fonctions.

time_test_PROG_DYN.py

- Ce fichier sert de fichier teste pour l'exécution et la validité des fonctions **DIST_1** et **SOL_1** à partir de **PROG_DYN**.

```
import PROG_DYN as prgd
import distance
import sys


if(len(sys.argv)==1):
    print("Erreur pas de fichier en argument ou chemin incorrect vers le fichier")
    exit()


file = open(sys.argv[1],"r")
lignes = file.readlines()
dist = distance.Struct_Instance(lignes)
print(prgd.prog_dyn(dist["X"],dist["Y"]))
file.close()
```

- Voici la courbe de consommation de temps CPU qu'on a pu obtenir à partir des instances résolut en moins de 10 minutes pour la fonction **PROG_DYN**.



Oui, la courbe obtenue correspond bien à la complexité théorique trouver précédemment. On peut apercevoir avec le graphe que la courbe est quadratique comme la complexité temporelle trouver dans les questions précédentes.

- La quantité mémoire utilisé par PROG_DYN pour l'instance Inst_0001000_23.adn est de 3.574 MB. On constate que PROG_DYN est plus efficace que la fonction DIST_NAIF_REC de la partie 1 puisqu'ici on exécute le problème pour une instance de grande taille mais la taille mémoire reste moins élevé que si on aurait exécuté cette instance avec DIST_NAIF_REC.

```
PS C:\Users\adcha\Desktop\Projet Algo> python -m memory_profiler .\test_prog_dyn.py .\Instances_genome\Inst_0001000_23.adn
((array([[   0,    2,    4, ..., 1770, 1772, 1774],
       [   2,    0,    2, ..., 1768, 1770, 1772],
       [   4,    2,    0, ..., 1766, 1768, 1770],
       ...,
       [1996, 1994, 1992, ...,  536,  538,  540],
       [1998, 1996, 1994, ...,  538,  540,  538],
       [2000, 1998, 1996, ...,  540,  542,  540]]), 540), ('CCCCACAGGGCTAGGGGT_AGTTCCCCTTCGGATTGTAGAGCCAGTACTTCACACTCACACTGGGACAATTAAGAGTATCCACAATTCTACATTGACTTCCCCCTGTC_GGATGGCGAGT
TGATTTATGTCCAAACCG_AAGGTACACTACCTTCGCAGAGTCTTGCGTTGTACCTACGTCTAAGCGTCCTCGAGAGTAAGGTGCAACCAATTATACACTCGGCGCCCACCGCACAAGCCGCGACATTAGCGCTTCGGAGC_GCAGTAGACTGTTGGTAACG_CTCGCAGT_ACGGTGCC
_GTCCGTG_AACTTGGATACTCCAAGCGCGGTTCCGAT_AGTTCGCTCCCGTGTGGTCCTTTACGCAC_GCGCGAACCTTAAATGTTGGGTAACACTCGATTGAAGAGTCAT_C_AGTTAAAGTAAAACTGTTAC_AAGGTCAGCCTCTCGTGAA_AAAATG_CAAGAGATCGCAATCCC
AGAGAGGAAATATAC_TACTGCCTTATT__TGGGGCAGCGCGGCAAGACAAACACATATGTCCTCCGCTCTGTTC_TTTTGCTAATCAGCTCAAATCC_AAGCTGGCTAGCTGAGGAATTGAC_TACCTA_ATAAAC_GGA__AAACTGGACGTGAAGGCGTAACA_C_CACAGTTGTTC
GATGCAATTTTAGATGTGGGC_ATCGCGGAATGACCCCGAGATGGTTCCATTCTCTTTAATTTATGCAT_T_ATTTAACTCTCACATTT_CCCCGGGGACTAAGATATTTAGGTTGGTTAGGCTCTATAAACTATTTTGCC_____GTCACTGACAGC_C__AAAATAATCTT_GGCTC
TGTCTTTGATTAAGGA_GGATGACCCA_TTCCG_GCACGTGCTAACAGTAGAAAGACTC_CGGTTCAGCCGGTTTTTGTTATAATAGCCT_A_TAACTCCTTTAAC_TTAGATA_AGAAAAACCTGGTGTCTAGCTGGGTAGA_TTATAACAACGCGCCATGCCGCTATAACAGC_TACG
AC_TTCGATCCGCG_CATCCCCACCCGCTA_CGG', 'CCCCACAGG_CTAGGG_TTACTTCCC_T_CGG_TTGTA_AGC_AG__CT_CACACTCACACTG_TACAATTAAG_GTATCC_CAAT_CT_CAT_CACT_CCCC_TGTCTG_ATGGCGCGT_GATT_ATGTC_AAACCGCA_G
GTACACTAC_T_CGCA_AGTCTTGCGTTGTACCTACGTCTAA_CG_CC_CGCGTG_ACGGAGCAACCAATTAT_CAATCGGCGCCCACCGCA_AAGCAGCGTC_TTAGCGCT_CG_A_CCGCAGTAGAC_GTTGGTAA_GGCTCGCA_TTACGGTGCCTG_CC_TGCA_CT_GGATA_TC
C__GCGCGGTTCCG_TTACT_C_CTCCCGTGTG_TCCTTT_CGC_CCGCGCGAA_GT_AAATGTTGGGTA_CTCTCGGT_GAA_ACTC_TTCTA_T_A__GTAAA_CT_T_GCCAAGGTCATCCTC_CGTGAACAAA__GGCAA_AGATC_CAATCC_A_ATAG_AAATCT_CCTACTGC
CTTATTCGTG__CAGCGCG_CAAGACAAACA_ATAT_TCCTCCGCTCTGT_CATTT_GCTAA_CAGCTCAA__CCCA_G_TGGC_AGCTG_GGA_T_G_CCTACC_AGA_AA_CCG_ACCA__TG_ACGTGAAGGCGTAACAGCTCA__GTTGT_CGA_GCAATTT__GA_GTGGGCC
ATCGCGGA_TGACCCCGCGATGGTTCCATACTC___AAT__A_G_ATATTATTGAACTC_CACATTTGCC__GGTTACTA_G_TATTTAGGT_GGTTCGGATCTATAAACTATTT_GCCTTAAAAGT_A_TG_C___C_CTAAA_T___CTTTG_CTCTGTCTT_GA_GAAG_ATG_AT_
_CC_AAT_C_GTG_ACGTGCTAACAGTA_AAAGACTCGCG_TTCTGCAG_TTTTTGTTATAATAGC_TTAATAA_T__T__A_CCTTAGATATA_AAAA_CCTGGTGTCTAGCTGGGTAGAGT_ATAACAACGCGC_ATGCCGATAT_TCA_CCTA_G_CCTTCGATCCG_GGCATCCCC
ACCC___AATG_'))
Filename: .\PROG_DYN.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     4   58.344 MiB   58.344 MiB           1   @profile
     5                                         def prog_dyn(x,y):
     6
     7   61.777 MiB    3.434 MiB           1       dist = DIST_1.dist_1(x,y)
     8   61.918 MiB    0.141 MiB           1       align = SOL_1.sol_1(x,y,dist[0])
     9
    10                                             #print("distance minimal entre x et y (DIST_1) : "+str(dist[1]))
    11                                             #print("alignement optimale entre x et y (SOL_1) : "+str(align))
    12
    13   61.918 MiB    0.000 MiB           1       return (dist,align)
```

3.3 Amélioration de la complexité spatiale du calcul de la distance

Question 19 :

Lors du remplissage de la ligne $i > 0$ du tableau T de **DIST_1**. Il nous suffirait d'avoir accès qu'aux lignes $i-1$ et i du tableau puisque lors de l'exécution de l'algorithme on ne compare que les cases $D[i-1][j-1], D[i][j-1], D[i-1][j]$ du tableau T.

Question 20 :

$DIST2(x,y)$:

Soit une matrice D de taille $2 \times (m+1)$

On initialise la matrice D tel que :

$D[0][j] = j \times C_{ins}, \forall j \in [0 \dots m]$ et $D[1][0] = C_{del}$

$k = 1$

Tant que $k \leq n$

Pour j allant de 1 à m faire:

$$D[1][j] = Min\left(D[1][j-1] + C_{ins}, D[0][j] + C_{del}, D[0][j-1] + C_{sub}(x_k, y_j)\right)$$

$k = k + 1$

On supprime la première ligne du tableau avec la seconde ligne.

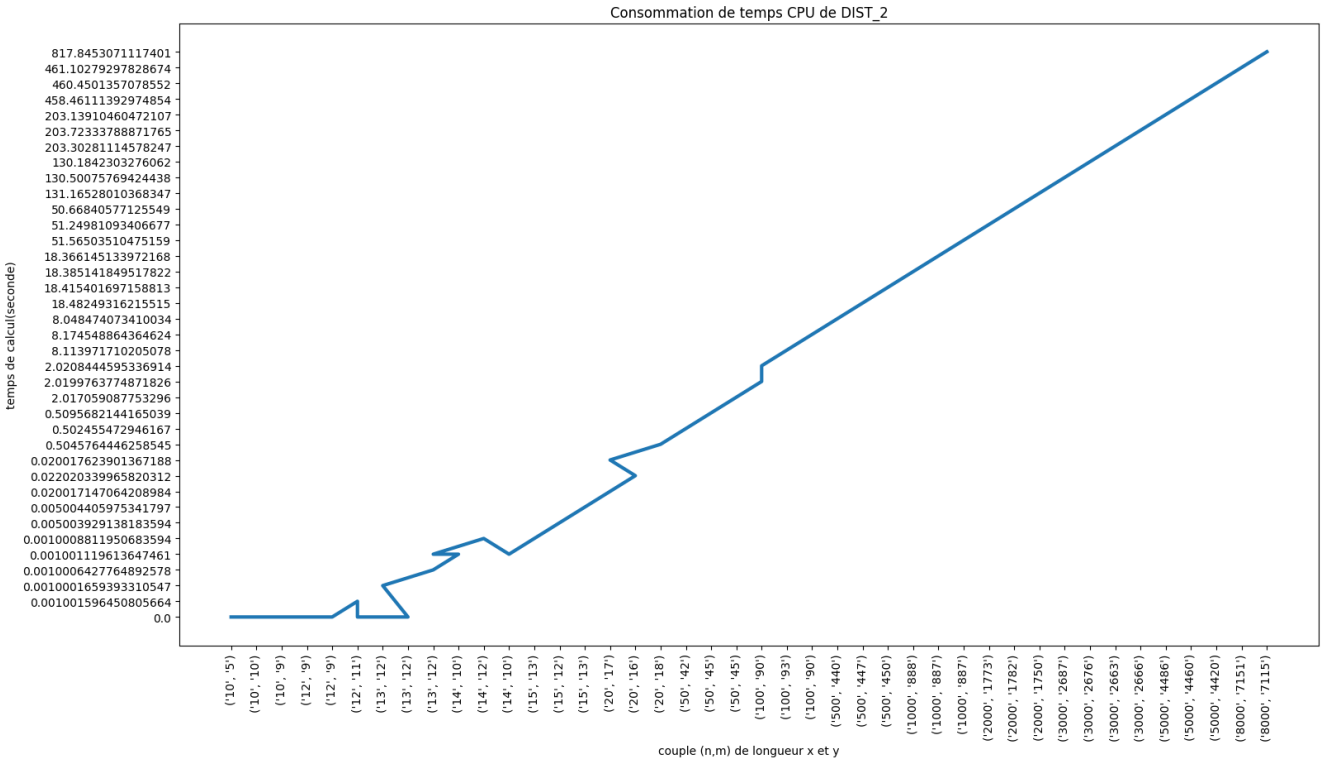
$D[1][0] = D[0][0] + C_{del}$

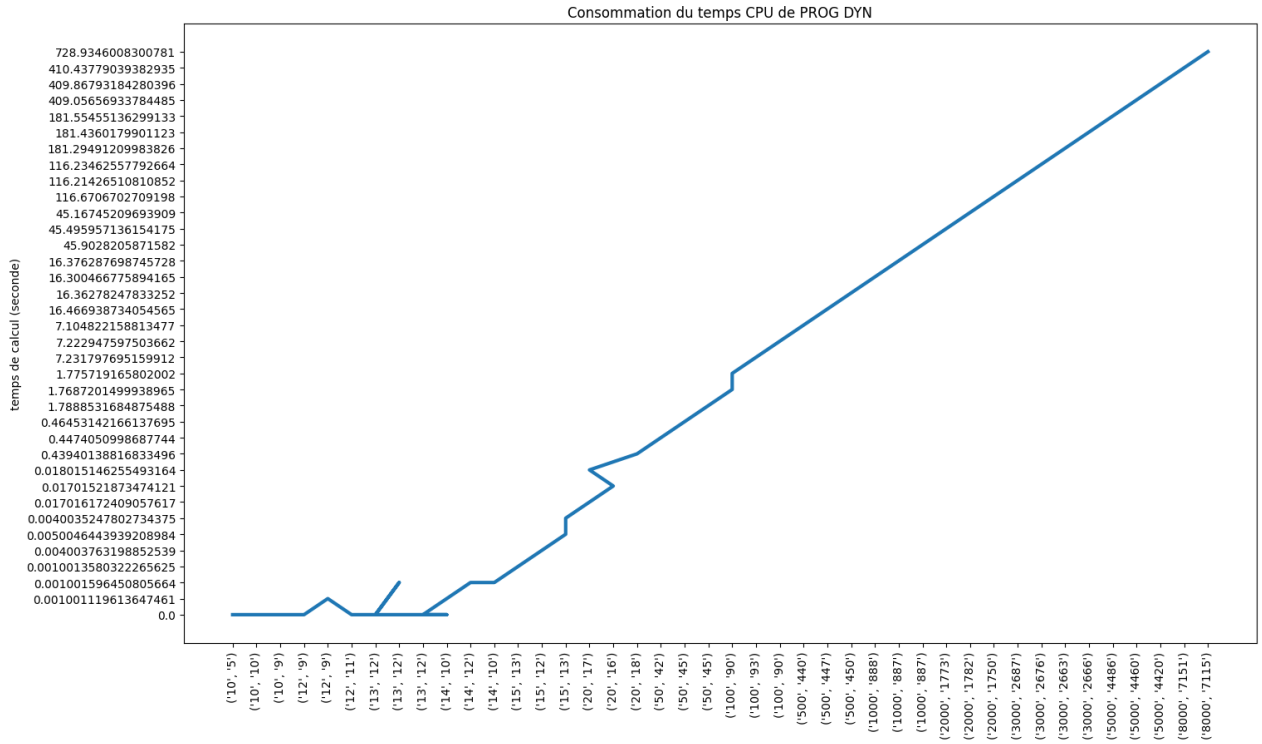
renvoie de $D[1][m]$

Tache C :

La fonction **Dist_2** est représenté ci-dessous.

```
import distance as dist
import numpy as np

def dist_2(x,y):

    n = len(x)+1
    m = len(y)+1
    D = np.zeros((2,m),dtype=int)
    k = 1

    for j in range(0,m):
        D[0][j]=j*dist.cins()

    D[1][0] = dist.cdel()

    for k in range(1,n):

        for j in range(1,m):
            D[1][j]=min(D[0][j-1]+dist.csub(x[k-1],y[j-1]),D[0][j]+dist.cdel(),D[1][j-
1]+dist.cins())

        for w in range(0,m):
            D[0][w]=D[1][w]

        D[1][0]=D[0][0]+dist.cdel()

    return D[0][m-1]
```





On peut constater d'après les graphiques ci-dessus que les fonctions DIST_2 et DIST_1 ont la même complexité temporelle qui est une complexité quadratique.

Pour une instance de très grande taille prenons par exemple 15000 caractères dans un fichier. On aura environ Longueur de x * taille d'un caractère = 15000*2 = 30000= 29.30 Kilooctets

Taille matrice = nombre de ligne* colonnes *octets = 2*taille de x * octets=2*15000*2 = 60000= 58,59 Kilooctets soit au total 29.30+58.59 = 87.89 Kilooctets. Ce qui se confirme avec la capture d'écran ci-dessous. On a une meilleure complexité mémoire que la fonction **DIST_1** précédente.

```
PS C:\Users\adcha\Desktop\Projet Algo> python -m memory_profiler .\test_DIST_2.py .\Instances_genome\Inst_0001000_23.adn
540
Filename: .\DIST_2.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     4   58.238 MiB   58.238 MiB           1   @profile
     5                                         def dist_2(x,y):
     6
     7   58.238 MiB    0.000 MiB           1       n = len(x)+1
     8   58.238 MiB    0.000 MiB           1       m = len(y)+1
     9   58.238 MiB    0.000 MiB           1       D = np.zeros((2,m),dtype=int)
    10   58.238 MiB    0.000 MiB           1       k = 1
    11
    12   58.238 MiB    0.000 MiB         889       for j in range(0,m):
    13   58.238 MiB    0.000 MiB         888           D[0][j]=j*dist.cins()
    14
    15   58.238 MiB    0.000 MiB           1       D[1][0] = dist.cdel()
    16
    17   58.246 MiB  -47.965 MiB        1001       for k in range(1,n):
    18
    19   58.246 MiB -42526.008 MiB      888000           for j in range(1,m):
    20   58.246 MiB -42478.129 MiB      887000               D[1][j]=min(D[0][j-1]+dist.csub(x[k-1],y[j-1]),D[0][j]+dist.cdel(),D[1][j-1]+dist.cins())
    21
    22   58.246 MiB -42597.094 MiB      889000           for w in range(0,m):
    23   58.246 MiB -42549.145 MiB      888000               D[0][w]=D[1][w]
    24
    25   58.246 MiB  -47.965 MiB        1000          D[1][0]=D[0][0]+dist.cdel()
    26
    27   58.145 MiB   -0.102 MiB           1       return D[0][m-1]
```

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

Question 21 :

$mot\_gaps(k)$:

x = ϵ (pour un mot k vide)
Pour i allant de 1 à k faire
$x = x.-$
Renvoie de x

Question 22 :

$aligne\_lettre\_mot(x, y)$:

Soit x un mot de longueur 1 et y un mot de longueur quelconque qu'on note m

On vérifie si y contient la lettre du mot x
Si oui on note sa position i
- on ajoute $i - 1$ gaps à gauche de x avec mot_gaps($i - 1$)
- on ajoute $m - i$ gaps à droite de x avec mot_gaps($m - i$)

Sinon
- On cherche une lettre dans y pour former une paire correspondante avec la lettre de x
- Si on la trouve, on note son indice i :
    - On ajoute $i - 1$ gaps à gauche de x
    - On ajoute $m - i$ gaps à droite de x
- Sinon on ajoute $m - 1$ gaps à droite de x

On retourne (x, y)

Question 23 :

Prenons $\overline{s} = BAL -, \overline{t} = RO -, \overline{u} = -LON$ et $\overline{v} = -ND - -$. Montrons que $(\overline{s}.\overline{u}, \overline{t}.\overline{v})$n'est pas un alignement optimal de (x, y).

En calculant le coût avec $\overline{s}.\overline{u} = BAL - -LON$ et $\overline{t}.\overline{v} = RO - -ND - -$, on a $C = C_{sub}(B, R) + C_{sub}(A, O) + C_{sub}(-, -) + C_{ins} + C_{sub}(L, D) + 3 \times C_{del}$ = 5+5+0+3+5+3*3 = 27.

Alors que pour un $\overline{t'} = R - -$ et $\overline{v'} = OND - -$, on a un coût pour $(\overline{s}.\overline{u}, \overline{t'}.\overline{v'})$ de 26. Donc $(\overline{s}.\overline{u}, \overline{t}.\overline{v})$ n'est pas un alignement optimal de (x, y).

Question 24 :

$SOL2(x, y)$:

- Si x est de longueur 1
  On retourne $aligne\_lettre\_mot(x, y)$

- Si y est de longueur 0
  On retourne $(x, mot - gaps(longueur\ de\ x))$

- Sinon
  $(x_1, y_1) = SOL2\left(x_{\left[1 \ldots \frac{n}{2}\right]}, y_{[1 \ldots j]}\right)$
  $(x_2, y_2) = SOL2\left(x_{\left[\frac{n}{2} \ldots n\right]}, y_{[j+1 \ldots m]}\right)$
  On retourne $(x_1.x_2, y_1.y_2)$ avec j la coupure (x, y) et n la longueur de x

Question 25 :

$COUPURE(x, y)$:
Soit une matrice D et une matrice I de taille $2 \times (m + 1)$

Initialisation de D tel que:
$D[0][j] = C_{ins} \times j$, $\forall j \in [0 \ldots m]$ et $D[1][0] = C_{del}$
On initialise I tel que :
$I[0][j] = j$, $\forall j \in [0 \ldots m]$ et $I[1][0] = I[0][0]$

$k = 1$ Et $m = longueur(y)$
Tant que $k \leq n$
Pour j allant 1 à m faire:

On place en $D[1][j]$ = $Min\left(D[1][j-1] + C_{ins}, D[0][j] + C_{del}, D[0][j-1] + C_{sub}(x_k, y_j)\right)$
Si $D[1][j] = D[0][j] + C_{del}$ alors $I[1][j] = I[0][j]$
Si $D[1][j] = D[1][j-1] + C_{ins}$ alors $I[1][j] = I[1][j-1]$
Si $D[1][j] = D[0][j-1] + C_{sub}(x_k, y_j)$ alors $I[1][j] = I[0][j-1]$

On supprime dans les deux matrices la première ligne avec la deuxième.
Donc $D[1][0] = D[0][0] + C_{del}$ et $I[1][0] = I[0][0]$

On renvoie $I[1][m]$

Question 26 :

La complexité spatiale de coupure est en $\theta(m)$ puisque l'algorithme utilise deux matrices de taille $2 \times (m+1)$.

Question 27 :

L'algorithme **SOL_2** décompose les problèmes de taille n par deux sous-problèmes chacune de taille $\frac{n}{2}$. On obtient à la fin de la résolution des sous-problèmes un arbre des appels récursifs de hauteur $\log(n)$. Au total, on réalise $k \times \log(n)$ appels récursifs avec $k > 0$. Au pire cas, pour chaque appel on appelle la fonction **coupure** qui est de complexité spatiale $\theta(m)$. Donc **SOL_2** est de complexité spatiale en $\theta(m \times \log(n))$.

Question 28 :

La complexité temporelle de **coupure** est en $\theta(n \times m)$ car **DIST_2** est composé de deux boucles imbriquées dans son algorithme. Donc il réalise $(n \times m)$ itérations avec pour chaque itération l'exécution d'opérations élémentaires.

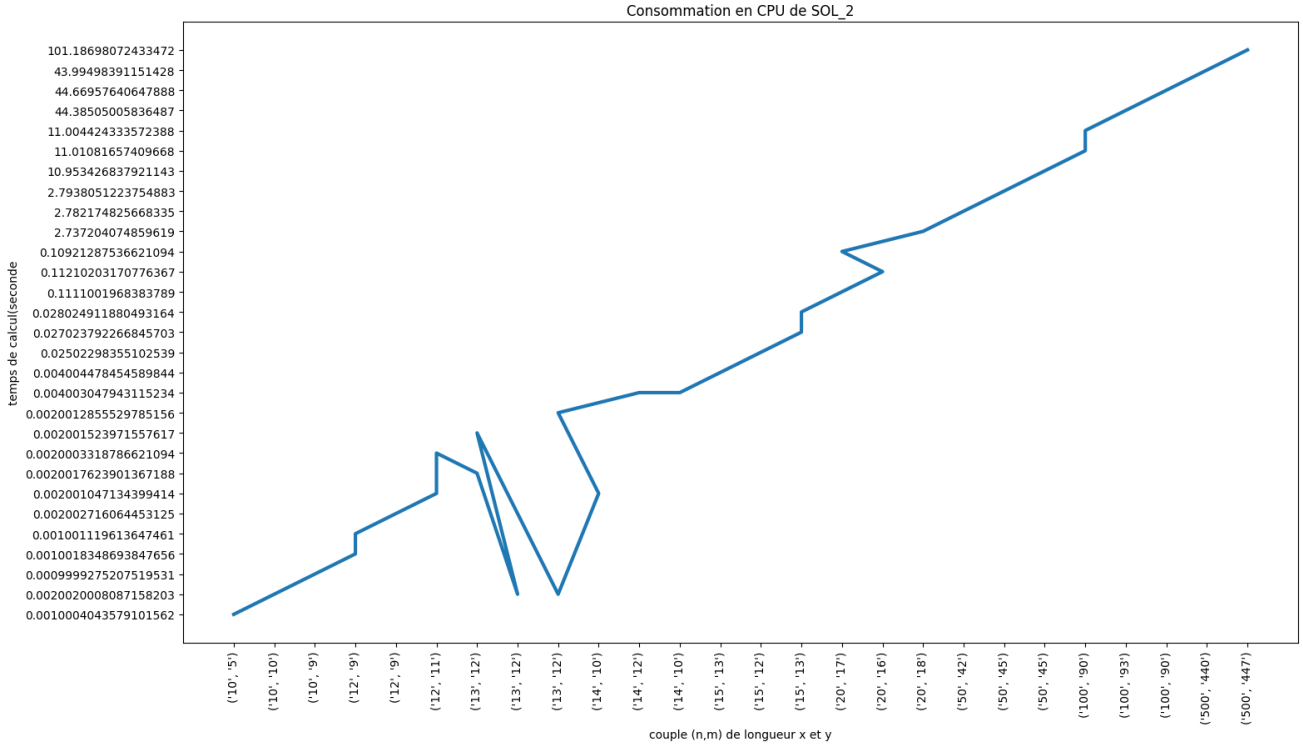
Tache D :

La fonction **SOL_2** est représenté ci-dessous et les autres fonctions sont présentes dans leurs fichiers respectifs à leurs noms.

```
import mot_gaps
import DIST_1
import aligne_lettre_mot
import coupure


def sol_2(x,y):

    xbis=''.join(x)
    ybis=''.join(y)

    n=len(xbis)
    m=len(ybis)

    if(n==1):
        return (aligne_lettre_mot.aligne_lettre_mot(xbis,ybis),ybis)
    if(m==1):
        return (x,aligne_lettre_mot.aligne_lettre_mot(ybis,xbis))
    if(n==0):
```

```
        return (mot_gaps.mot_gaps(m),ybis)
    if(m==0):
        return (x,mot_gaps.mot_gaps(n))
    else:

        w = coupure.coupure(x,y,DIST_1.dist_1(xbis,ybis)[0])

        u=sol_2(xbis[:int((n/2))],ybis[:int(w)])
        v=sol_2(xbis[int((n/2)):],ybis[int(w):])


        return (u[0]+v[0],u[1]+v[1])
```

Voici la courbe de calcul CPU obtenu avec cette fonction pour un certains nombre d'instances donnée.



Voici la consommation mémoire de cet algorithme sur l'instance Inst_0001000_23.adn.

```
PS C:\Users\adcha\Desktop\Projet Algo> python -m memory_profiler .\test_SOL_2.py .\Instances_genome\Inst_0001000_23.adn
('CCCCACAGGGCTAGGGGTAGTTCCCCTTCGGATTGTAGAGCCAGTACTTCACACTCACACTGGGACAATTAAGAGTATCCACAATTCTACATTGACTTCCCCCTGTCGGATGGCGAGTTGATTTATGTCCAAACCGAAGGTACACTACCTTCGCAGAGTCTTGCGTTGTACCTACGTCTAAGCGTCCTCGAGAGT
AAGGTGCAACCAATTATACACTCGGCGCCCACCGCACAAGCCGCGACATTAGCGCTTCGGAGCGCAGTAGACTGTTGGTAACGCTCGCAGTACGGTGCCGTCCGTGAACTTGGATACTCCAAGCGCGGTTCCGATAGTTCGCTCCCGTGTGGTCCTTTACGCACGCGCGAACCTTAAATGTTGGGTAACACTCGATT
GAAGAGTCATCAGTTAAAGTAAAACTGTTACAAGGTCAGCCTCTCGTGAAAAAATGCAAGAGATCGCAATCCCAGAGAGGAAATATACTACTGCCTTATTTGGGGCAGCGCGGCAAGACAAACACATATGTCCTCCGCTCTGTTCTTTTGCTAATCAGCTCAAATCCAAGCTGGCTAGCTGAGGAATTGACTACCTA
ATAAACGGAAAACTGGACGTGAAGGCGTAACACCACAGTTGTTCGATGCAATTTTAGATGTGGGCATCGCGGAATGACCCCGAGATGGTTCCATTCTCTTTAATTTATGCATTATTTAACTCTCACATTTCCCCGGGGACTAAGATATTTAGGTTGGTTAGGCTCTATAAACTATTTTGCCGTCACTGACAGCCAAA
ATAATCTTGGCTCTGTCTTTGATTAAGGAGGATGACCCATTCCGGCACGTGCTAACAGTAGAAAGACTCCGGTTCAGCCGG__T_TTTTG_T_TATAATAGCCTATAACTCCTTTAACTTAGATAAGAAAAACCTGGTGTCTAGCTGGGTAGATTA___TAACAACGCGCCATGCCGCTATAACAGCTACGACTTCG
ATCC__GCGC________A_T_CCCCA__C_CC_GC__T_______A_C_____GG_____________________________________________________________________________________________________________________________________________
_____________________________________________________________________________________________________________________________________________________________________________________________________
_____________________________________________________________________________________________________________________________________________________________________________________________________
_____________________________________________________________________________________________________________________________________________________________________________________________', 'C_____
_____________________________________________________________________________________________________________________________________________________________________________________________________
_____________________________________________________________________________________________________________________________________________________________________________________________________
_____________________________________________________________________________________________________________________________________________________________________________________________________
_______C_____________________________________________________________________________________________________________________________________________________________________________________________
_____________________________________________________________________________________________________________________________________________________________________________CCA__C__AGG_____
_____________________________C_____________TAGGGT_____________T_____________ACTTC_CCTCGGTTGTAA__GC_AGCTCACACTCACACTGTACAATTAAGGTATCCCAATCTCATCACTCCCCTGTCTGATGGCGCGTGATTATGTCAA
ACCGCAGGTACACTACTCGCAAGTCTTGCGTTGTACCTACGTCTAACGCCCGCGTGACGGAGCAACCAATTATCAATCGGCGCCCACCGCAAAGCAGCGTCTTAGCGCTCGACCGCAGTAGACGTTGGTAAGGCTCGCATTACGGTGCCTGCCTGCACTGGATATCCGCGCGGTTCCGTTACTCCTCCCGTGTGTCC
TTTCGCCCGCGCGAAGTAAATGTTGGGTACTCTCGGTGAAACTCTTCTATAGTAAACTTGCCAAGGTCATCCTCCGTGAACAAAGGCAAAGATCCAATCCAATAGAAATCTCCTACTGCCTTATTCGTGCAGCGCGCAAGACAAACAATATTCCTCCGCTCTGTCATTTGCTAACAGCTCAACCCAGTGGCAGCTGG
GATGCCTACCAGAAACCGACCATGACGTGAAGGCGTAACAGCTCAGTTGTCGAGCAATTTGAGTGGGCCATCGCGGATGACCCCGCGATGGTTCCATACTCAATAGATATTATTGAACTCCACATTTGCCGGTTACTAGTATTTAGGTGGTTCGGATCTATAAACTATTTGCCTTAAAAGTATGCCCTAAATCTTTG
CTCTGTCTTGAGAAGATGATCCAATCGTGACGTGCTAACAGTAAAAGACTCGCGTTCTGCAGTTTTTGTTATAATAGCTTAATAATTACCTTAGATATAAAAACCTGGTGTCTAGCTGGGTAGAGTATAACAACGCGCATGCCGATATTCACCTAGCCTTCGATCCGGGCATCCCCACCCAATG')
Filename: .\SOL_2.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     5   58.605 MiB   56.020 MiB          83   @profile
     6                                         def sol_2(x,y):
     7
     8   58.605 MiB   -2.277 MiB          83       n=len(x)
     9   58.605 MiB   -2.277 MiB          83       m=len(y)
    10                                             #cas de bases lors que longueur de x=1 ou y=0
    11   58.605 MiB   -2.277 MiB          83       if(n==1):
    12   58.605 MiB   -0.949 MiB          22           return (aligne_lettre_mot.aligne_lettre_mot(x,y),y)
    13   58.605 MiB   -1.246 MiB          61       if(m==1):
    14   58.605 MiB   -0.180 MiB          11           return (x,aligne_lettre_mot.aligne_lettre_mot(y,x))
    15   58.605 MiB   -1.066 MiB          50       if(n==0):
    16                                                 return (mot_gaps.mot_gaps(m),y)
    17   58.605 MiB   -1.066 MiB          50       if(m==0):
    18   58.602 MiB   -0.008 MiB           9           return (x,mot_gaps.mot_gaps(n))
    19                                             #recursivte de l'algorithme créer les sous-appels de taille n/2
    20                                             else:
    21
    22   58.605 MiB   -0.836 MiB          41           w = coupure.coupure(x,y,DIST_1.dist_1(x,y)[0])
    23
    24   58.605 MiB   -1.125 MiB          41           u=sol_2(x[:int((n/2))],y[:int(w)])
    25   58.613 MiB   -0.793 MiB          41           v=sol_2(x[int((n/2)):],y[int(w):])
    26
    27   58.613 MiB   -0.781 MiB          41           return (u[0]+v[0],u[1]+v[1])
```

Question 29 :

Oui, l'amélioration de la complexité spatiale nous a fait perdre en complexité temporelle. La comparaison expérimentale montre cette différence.