**Algorithme anti-plagiat – Applied Algorithms**

***Notes*** :

*Nous avons choisi d’utiliser python (bien qu’interprété). L’énoncé demandant un rendu compilable et cela étant possible avec python, nous avons préféré nous concentrer sur la partie algorithmique en utilisant un langage que nous maîtrisons mieux que le C.*

*Pour suivre l’évolution de notre projet : git clone* [*https://github.com/franckdeturchedura/Detection\_plagiat.git*](https://github.com/franckdeturchedura/Detection_plagiat.git)

**Question 1 :**

Notre but est de trouver le score d’alignement optimal (c’est-à-dire un **score d’alignement minimisé)** de deux séquences données x et y (suites de caractères). Le tout doit être fait par un algorithme en **O(|x| \* |y|).**

Pour cela nous allons reprendre le principe d’un algorithme connu du cours : **Edit String**. Celui-ci a pour but, initialement, de donner le nombre d’opérations nécessaires pour obtenir, à partir de deux séquences différentes, des séquences identiques.

Les dites opérations autorisées sont Ins(), Del() et Sub(x,y) qui permettent respectivement d’obtenir le coût de l’insertion d’un caractère, le coût de la suppression d’un caractère et le coût de l’échange de caractères.

Pour calculer le score optimal d’alignement, on affecte à Del() et Ins() un coût de +1 (car plus le score est haut, pire est l’alignement). Pour Sub(), nous n’affectons aucun coût si les lettres des deux séquences sont les mêmes. Nous affectons un coût de +1 dans le cas contraire.

Ainsi, nous obtenons une matrice telle que pour tout i et tout j, t[i][j] donne la distance entre les séquences xi, yj, préfixes de x et y de longueurs i et j.

**Pseudo Code :**

**Données :**

* Séquence A : A
* Séquence B : B
* Del() : retourne 1
* Ins() : retourne 1
* Sub(x,y) : retourne 0 si x = y et 1 sinon

**distance(A,B)** {

lenA = longueur(A) #Taille de la séquence A

lenB = longueur(B) #Taille de la séquence B

T[0][0] = 0

Pour i allant de 1 à lenA :

T[i][0] = T[i-1][0] + Del()

Pour j allant de 1 à lenB :

T[0][j] = T[0][j-1] + Ins()

Pour i allant de 1 à lenA :

Pour j allant de 1 à lenB :

T[i][j] = min(

T[i][j-1] + Ins(),

T[i-1][j] + Del(),

T[i-1][j-1] + Sub(A[i-1], B[j-1])

)

**retourner T**

}

Comme pour Edit String, on a une complexité en lenA \* lenB (soit |x|\*|y|).

**Question 2 :**

Via l’algorithme précédent, nous obtenons pour deux séquences données x et y, la matrice t de taille (|x|+1)\*(|y|+1), telle que t[i][j] est le score d’alignement optimal entre xi et yj.

Nous cherchons maintenant à obtenir, à partir de deux séquences données et de l’algorithme précédent, l’alignement optimal x’,y’ des deux séquences x et y.

L’algorithme doit être de **complexité linéaire**.

Tout d’abord, nous avons implémenté une fonction retournant, à partir de deux indices donnés et la matrice T renvoyée par l’algorithme ci-dessus, les indices suivants à utiliser pour « remonter » la matrice T et reconstituer x’ et y’.

Soient x et y les deux séquences initiales.

Soit m = |x| et n = |y| et T la matrice associée aux séquences x et y.

En effet, pour trouver l’alignement optimal x’,y’, nous allons partir du score d’alignement optimal de Xm, Yn (soit les deux séquences entières), à T[m][n]. Nous allons ensuite remonter la matrice en prenant les scores d’alignements minimaux (et donc optimaux) et en déduire les caractères de x’ et y’.

L’algorithme retourne basiquement les indices des valeurs minimales de la matrice à partir d’un point donné. Pour cela il vérifie pour chaque cas les valeurs des 3 cases accolées.

**Données :**

* indice i de départ
* indice j de départ
* Matrice des distances T

**retourne\_indices(i, j, T)** {

Si T[i-1][j] < T[i][j-1] et T[i-1][j] < T[i-1][j-1] :

i = i-1

**retourner i, j**

Sinon Si T[i][j-1] < T[i-1][j] et T[i][j-1] < T[i-1][j-1] :

j = j-1

**retourner i, j**

Sinon Si T[i-1][j-1] < T[i][j-1] et T[i-1][j-1] < T[i-1][j] :

i = i-1

j = j-1

**retourner i, j**

Sinon :

i = i-1

j = j-1

**retourner i, j**

}

Une fois cela fait, nous pouvons implémenter l’algorithme d’alignement.

Pour cela, nous faisons, tant que nous ne sommes pas revenus au point initial (T[0][0]), des conditions sur la différence entre les indices actuels et les indices suivants calculés via l’algorithme ***retourne\_indices()***. Une fois cela fait, nous avons la « direction » à prendre et n’aurons plus qu’à inverser les tableaux obtenus à la fin.

Ainsi, selon le résultat obtenu avec ces différences, on sait si, pour x’ et y’, on doit ajouter une lettre de la séquence de base ou étirer le texte avec un espace.

Les détails sont dans les commentaires de l’algorithme (position suivante horizontale et verticale -> tiret pour x’ ou y’ et lettre pour l’autre, position suivante diagonale -> lettres pour x’ et y’).

L’algo permettant de déterminer les indices suivants ayant une complexité constante, l’algo principal est bien de complexité linéaire.

Après tests sur différents textes et séquences de notre invention, l’algo retourne bien x’ et y’, cohérents et chacun pouvant être étirés.

**Données** :

* Séquence A : A
* Séquence B : B
* retourne\_indices(i,j,T) : fonction prenant i et j les indices actuels ainsi que T la matrice des distances. Retourne les indices suivants.
* longueur(a) : retourne la longueur de la string a
* distance(A,B) : retourne la matrice T

**alignements\_optimaux(A,B)** {

A\_p = [] # là où nous stockerons A’

B\_p =[] # là où nous stockerons B’

i = longueur(A) # pour partir de la dernière ligne de la matrice T

j = longueur(B) # pour partir de la dernière colonne de la matrice T

i\_p = 0 # contiendra la ligne suivante

j\_p = 0 # contiendra la colonne suivante

T = distance(A,B) # On instancie la matrice des scores optimaux

Tant que i différent de 0 et j différent de 0 : #tant qu’on n’est pas revenu au point de départ

i\_p, j\_p = retourne\_indices(i,j,T) #stocke dans i\_p et j\_p les indices suivants

Si i\_p – i =-1 ET j\_p -j =-1 Alors {  # Cas où le score optimal suivant est sur la diagonale

A\_p.append(A[i\_p]) # On ajoute à A’ la lettre correspondante de la séquence A

B\_p.append(B[j\_p]) # On ajoute à B’ la lettre correspondante à la séquence B

i = i-1 # on met à jour l’indice des lignes

j = j-1 # on met à jour l’indice des colonnes

}

Sinon si i\_p-i =-1 ET j\_p-j=0 Alors { # Cas où le score optimal suivant est sur la ligne

A\_p.append(A[i\_p])

B\_p.append(«  ») # on ajoute un espace (étirement) à B’

i = i-1 # on met à jour l’indice des lignes

}

Sinon Si i\_p-i =0 ET j\_p-j=-1 Alors { # Cas où le score optimal suivant est sur la colonne

A\_p.append(«  »)

B\_p.append(B[j\_p])

j = j-1 # on met à jour l’indice des colonnes

}

inverser(A\_p) # On remet les séquences à l’endroit car on a parcouru les textes à l’envers

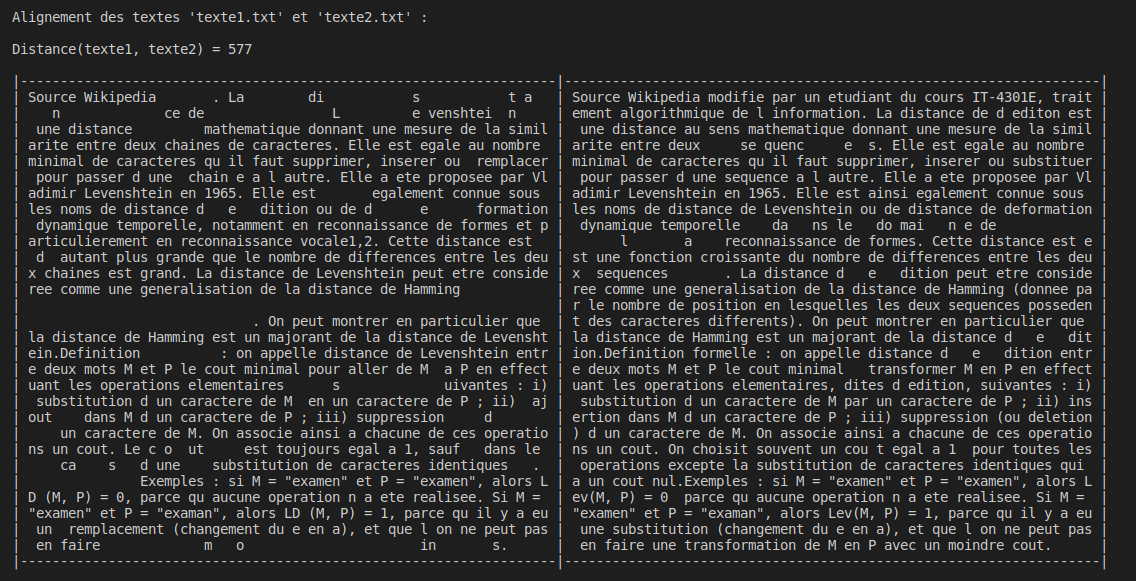
inverser(B\_p)

**retourner A\_p, B\_p**

}

**Question 3 :**

On applique la fonction ***alignements\_optimaux()*** aux fichiers ‘texte1.txt’ et ‘texte2.txt’ :



**Question 4 :**

L’inconvénient de cette implémentation dans le cadre de la détection de plagiat dans un texte est qu’elle ne peut aligner correctement qu’un seul paragraphe ou ligne là où la plupart des textes sont organisés en plusieurs paragraphes.

On cherche donc à améliorer l’algorithme pour pouvoir détecter un plagiat entre plusieurs paragraphes d’un texte et pour cela on va utiliser le fait que l’algorithme de Levensthein peut être utiliser pour calculer la distance qui sépare autre chose que des caractères comme par exemple des images ou bien des paragraphes entiers dans notre cas.

On va donc modifier la fonction ***distance()*** en définissant une nouvelle distance, la distance entre les paragraphes de deux textes. L’algorithme ***distance()*** reste inchangé, ce sont les coûts des différentes opérations qui vont changer.

***Del(seq)*** *et* ***Ins(seq)*** prennent maintenant une paragraphe en paramètre et le coût de suppression ou d’insertion de ce paragraphe correspond à la longueur de ce paragraphe.

Le coût d’une permutation donné par ***Sub(seq1, seq2)*** correspond à la distance entre les paragraphes seq1 et seq2.

Ainsi la fonction ***distance()*** est modifiée comme suit :

**Données :**

* Texte 1 T1
* Texte 2 T2
* Del2(seq) : retourne len(seq)
* Ins2(seq) : retourne len(seq)
* Sub2(x,y) : retourne distance(x, y)

**distance2(T1, T2)** {

lenT1 = longueur(T1)

lenT2 = longueur(T2)

T[0][0] = 0

Pour i allant de 1 à lenT1 :

T[i][0] = T[i-1][0] + Del2(T1[i])

Pour j allant de 1 à lenT2 :

T[0][j] = T[0][j-1] + Ins2(T2[j])

Pour i allant de 1 à lenT1 :

Pour j allant de 1 à lenT2 :

T[i][j] = min(

T[i][j-1] + Ins2(T2[j-1]),

T[i-1][j] + Del2(T1[i-1]),

T[i-1][j-1] + Sub2(T1[i-1], T2[j-1])

)

**retourner T**

}

La fonction ***alignement\_paragraphes(T1, T2)*** permettant de faire correspondre les paragraphes de T1 et T2 est identique à ***alignements\_optimaux(A, B)*** à ceci près qu’on utilise cette fois ***distance2()***.

Cette algorithme permet cette fois d’apparier les paragraphes plagiés entre deux textes.

Exemple sur les fichiers ‘t1.txt’ et ‘t2.txt’ :



Le score de similarité correspond au nombre de caractères en commun entre les deux textes, ramené au nombre total de caractères entre les deux textes.

**Fonctions d’affichage**

Les fonctions utiliser pour afficher les textes sont écrites en C dans le fichier *affichage.c* et sont importées par le fichier *affichage.so* dans Python à l’aide du packages *ctypes.*

Pour compiler le fichier *affichage.so*, il faut lancer lacommande bash :

*gcc -shared -Wl,-soname,affichage -o affichage.so -fPIC affichage.c*

**Instructions d’exécution**

La marche à suivre pour reproduire le fichier TD3.c ainsi que l’exécutable, voir le fichier README.md ou le projet Github donné en début de rapport.

Pour exécuter le fichier compilé, lancer la commande bash :

*./TD3*