**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 plutôt que sur la difficulté d’utilisation d’un langage que nous n’utilisons jamais en DSIA (C ou JAVA dont nous sommes donc peu familiers)*

*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 sont Ins(), Del() et Sub(x,y) qui permettent respectivement d’obtenir le coût de l’insertion dans la première séquence d’ un caractère de la seconde, de la suppression d’un caractère ou de l’échange de caractères.

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

Ainsi, nous obtenons, au lieu d’une matrice donnant le nombre d’opérations pour obtenir deux séquences identiques telle que t[i][j] soit le nombre d’opérations pour les séquences xi,yj , une matrice telle que t[i][j] soit le score d’alignement optimal de xi,yj.

**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

**ScoreOpti(A,B)**

{

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

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

T = [] #on va initialisé le tableau à 0

Pour i allant de 0 à lenA :

Pour j allant de 0 à lenB :

T[i][j] = 0

Pour i allant de 1 à lenA : # même principe que Edit String

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

Pour j allant de 1 à lenB :

T[0][j] = T[0][j-1] + Ins() #même principe que Edit String

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’algo 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’algo 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 citée 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 (optimaux par conséquent) et en déduire les caractères de x’ et y’.

L’algo 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.

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

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’algo précédent. Une fois cela fait, nous avons la « direction » à prendre et n’aurons plus qu’à inverser les tableaux obtenus.

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 un tiret (« \_ », soit un étirement).

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.

**Pseudo-code**

**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 issue de l’algorithme précédent. Retourne les indices suivants
* inverse\_tab(T) : fonction prenant un tableau en entrée et retournant son inverse (i’ = -i)
* longueur(a) : retourne la longueur de la string a
* ScoreOpti(A,B) : fonction de la question 1

**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 = ScoreOpti(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 séquenceA

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

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]) # on ajoute à A’ la lettre correspondante de la séquenceA

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

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

}

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

{

A\_p.append(« \_ ») # on ajoute un tiret (étirement) à A’

B\_p.append(B[j\_p]) # on ajoute à B’ la lettre correspondante de la séquenceB

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

}

}

**Retourner inverse\_table(A\_p), inverse\_table(B\_p)** # comme on a remonté la matrice à l’envers, il faut remettre dans le bonne ordre les caractères de A’ et B’

}

**Question 3 :**

Cf ZIP