TP5 – DISTANCE D'EDITION - LEVENSHTEIN PRESENTATION

La distance de Levenshtein est une distance mathématique entre deux mots. Elle correspond au nombre minimal de caractères qu'il faut supprimer, ajouter ou remplacer pour passer d'un mot à l'autre. Elle a été introduite par Vladimir Levenshtein en 1965 et est aussi connue sous le nom de distance d'édition ou de déformation dynamique temporelle.
Les applications sont diverses, en voici des exemples :

- Correction orthographique
- Reconnaissance de formes
- Reconnaissance vocale

Amusez-vous à demander à Chat GPT s'il utilise cette distance.

Exemples de distances :

examen VS examan	mon VS ma	niche VS chien
examen examan	mon mn ma	niche iche che chie chien
Une substitution	Une suppression Une substitution	2 suppressions 2 ajouts
D=1	D=2	D=4
Il existe plusieurs chemins (suites) menant au même résultat avec la même distance minimale		

Présentation de l'algorithme

Soient deux chaines de caractères M_1 et M_2 de longueurs respectives N_1 et N_2. On appelle M_{1_0} la première lettre de M_1 et M_1^i le mot restant à partir de M_1 en lui ayant enlevé i lettres ($M_1 = M_{1_0} + M_1^1$), de même pour M_2^j. On trouve la distance de Levenshtein $lev(M_1, M_2)$ par la méthode descendante récursive suivante :

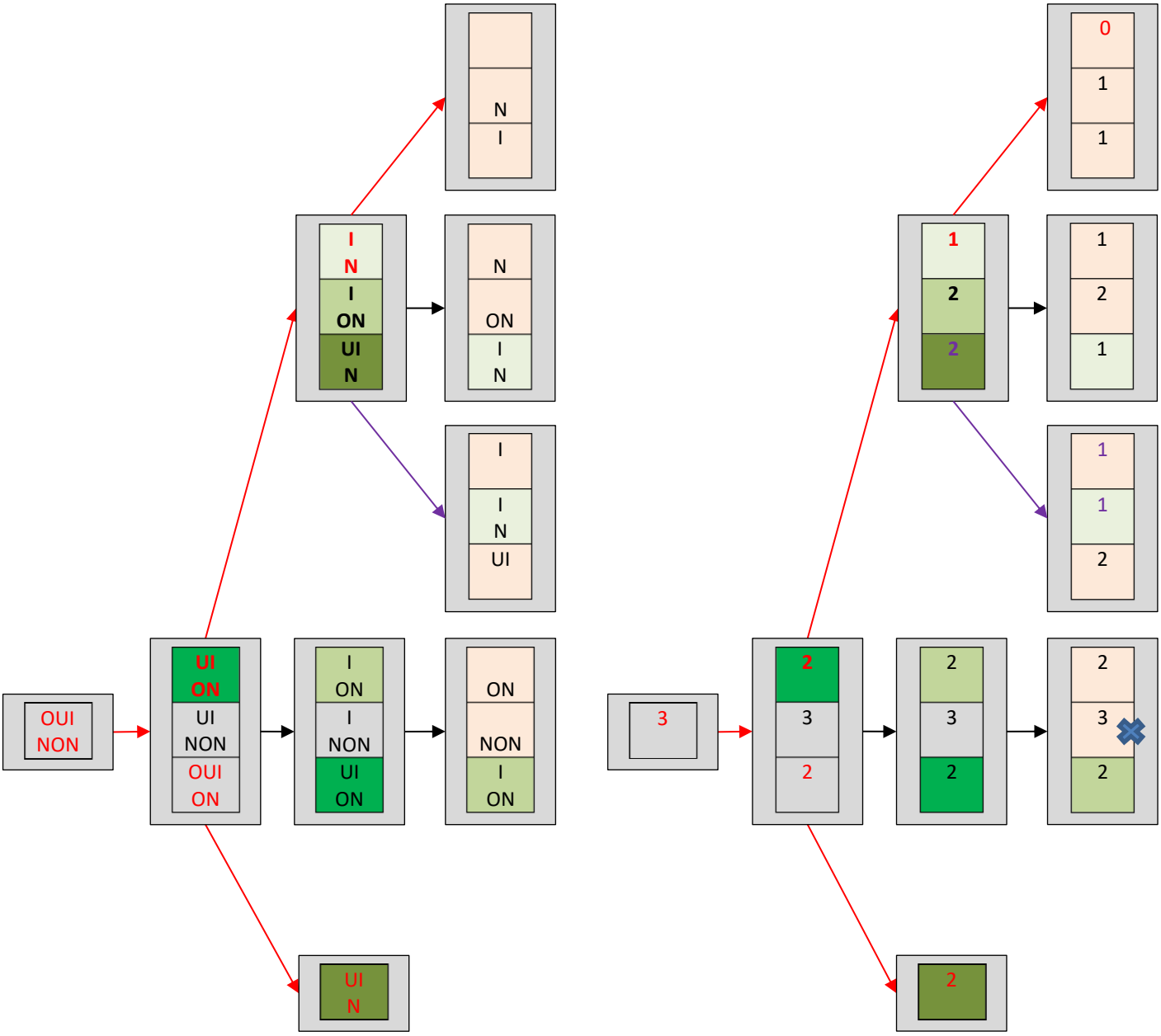
$$lev(M_1, M_2) = \begin{cases} max(N_1, N_2) \ si \ min(N_1, N_2) = 0 \\ lev(M_1^1, M_2^1) \ si \ M_{1_0} = M_{2_0} \\ 1 + min \begin{cases} lev(M_1^1, M_2^1) \\ lev(M_1^1, M_2) \ sinon \\ lev(M_1, M_2^1) \end{cases} \end{cases}$$

Illustration de l'algorithme

Déterminons la distance de Levenshtein entre OUI et NON.

Dans l'exemple ci-dessous :

- On ne recalcule volontairement pas les cas déjà rencontrés (verts) et on indique en gras la case ayant permis de trouver la valeur associée à une couleur verte
- On fait apparaître en orangé les cases « cas de base ». On propose deux arbres, celui de l'exécution, et celui des résultats de l'algorithme



La distance de Levenshtein entre OUI et NON est donc de 3.
On remarque que :

- Beaucoup de position sont calculées plusieurs fois, cet algorithme n'est donc pas optimisé. Nous l'optimiserons par mémoïsation et par un algorithme ascendant abordé plus tard
- Dans toutes les cases, avec comme présenté précédemment, M_1^i le mot restant à partir de M_1 en lui ayant enlevé i lettres (de même pour M_2^j), on remarque que pour un couple (i, j), il n'y a qu'un unique couple $\left(M_1^i, M_2^j\right)$. Plus concrètement, quand il y a par exemple 2 lettres dans le mots du dessus (1° enlevée), et 2 dans le mot du dessous (première enlevée), les mots sont forcément $M_1^1 = UI$ et $M_2^1 = ON$. On voit apparaître la possibilité de représenter toutes ces possibilités dans une table à deux dimensions, principe de l'algorithme ascendant.
- La case avec une croix bleu est le pire des cas de l'algorithme de transformation proposé : 3 suppressions suivies de 3 ajouts ou inversement

On a représenté en rouge et violet les transformations possibles avant une distance minimale de 3. On peut retrouver la succession des choix réalisés en trouvant les chemins menant aux minimums choisis. Lorsqu'il existe plusieurs chemins, on parle de bifurcation. Dans l'exemple ci-dessus, voici des chemins possibles :

- Haut : OUI/NON → UI/ON → I/N → _/_ (1)
- Bas 1 : OUI/NON → OUI/ON → UI/N → I/N → _/_ (2)
- Bas 2 : OUI/NON → OUI/ON → UI/N → I/_ (3)

Attention, chaque chemin peut être interprété selon que l'on modifie OUI pour devenir NON ou NON pour devenir OUI et n'est pas directement ce qui est écrit. Je note ci-dessous A pour Ajout, R pour Retrait, S pour Substitution et 0 pour rien :

Ch.	Trans.	Départ	Etape 1		Etape 2		Etape 3		Etape 4	
(1)	OUI/NON → UI/ON → I/N → _/_									
	OUI	OUI/NON	S	NUI/NON	S	NOI/NON	S	NON/NON		
		Bilan	OUI → NUI → NOI → NON							
	NON	OUI/NON	S	OUI/OON	S	OUI/OUN	S	OUI/OUI		
		Bilan	NON → OON → OUN → OUI							
(2)	OUI/NON → OUI/ON → UI/N → I/N → _/_									
	OUI	OUI/NON	A	NOUI/NON	0	NOUI/NON	R	NOI/NON	S	NON/NON
		Bilan	OUI → NOUI → NOI → NON							
	NON	OUI/NON	R	OUI/ON	0	OUI/ON	A	OUI/OUN	S	OUI/OUI
		Bilan	NON → ON → OUN → OUI							
(3)	OUI/NON → OUI/ON → UI/N → I/_									
	OUI	OUI/NON	A	NOUI/NON	0	NOUI/NON	S	NONI/NON	R	NON/NON
		Bilan	OUI → NOUI → NONI → NON							
	NON	OUI/NON	R	OUI/ON	0	OUI/ON	S	OUI/OU	A	OUI/OUI
		Bilan	NON → ON → OU → OUI							

On remarque :

- Que chaque ligne présente 3 opérations parmi A, S et R (distance de 3)
- Quand chacun des mots perd sa première lettre, c'est une substitution
- Quand un seul mot perd une lettre, c'est un retrait pour ce mot ou un ajout pour l'autre

Méthode en programmation dynamique

Wagner et Fischer ont proposé en 1974 une version en programmation dynamique de type « bas en haut » en traitant le sous problème suivant :

Quelle est la distance de Levenshtein entre $M1_i$ et $M2_j$, contenant respectivement les i premières lettres de M_1 et les j premières lettres de M_2

Voici la description de l'algorithme ascendant proposé :

$$\forall i \in [0, N_1], \forall j \in [0, N_2], D(i,j) = \begin{cases} i \text{ si } j = 0 \\ j \text{ si } i = 0 \\ \text{sinon} \begin{cases} D(i-1, j-1) \text{ si } M_1[i-1] = M_2[j-1] \\ 1 + min \begin{cases} D(i-1, j-1) \\ D(i-1, j) \quad \text{sinon} \\ D(i, j-1) \end{cases} \end{cases} \end{cases}$$

$D(N_1, N_2)$ contient le nombre minimal de modifications à réaliser pour passer de M_1 à M_2 (et inversement)

On introduit la table suivante :

	j	0	1	2	3	4	5
i			C	H	I	E	N
0		0	1	2	3	4	5
1	N	1					
2	I	2					
3	C	3					
4	H	4					
5	E	5					

La première ligne et la première colonne représentent les indices i et j de parcours des chaines CHIEN et NICHE, mais attention, à partir d'un indice 0 avant de commencer effectivement les premières lettres à l'indice 1.

Les cases violettes représentent l'initialisation, c'est-à-dire le nombre de modifications pour passer d'une chaine de caractères vide ($i = j = 0$) à la chaine de caractères :

- Pour $j = 0$: N (1 ajout), NI (2 ajouts) etc. jusqu'à NICHE (5 ajouts)
- Pour $i = 0$: C (1 ajout) , CH (2 ajouts) etc. jusqu'à CHIEN (5 ajouts)

On note $D(i,j), i \in [0, N_1]$, $j \in [0, N_2]$ la valeur dans la case du tableau à la ligne i et la colonne j et M_1, M_2 les mots tels que $M_1[i_l], i_l = i - 1$ pour $i \in [1, N_1]$ est une lettre de M_1 (de même pour $M_2[j_j]$ dans M_2). $M_1[i_l]$ et $M_2[j_l]$ sont donc les lettres en face de $D(i,j)$. On procède alors au remplissage de la sorte : Ayant rempli une case $D(i-1, j-1)$, on passe à la case suivante $D(i,j)$ et :

- Si $M_1[i_l] = M_2[i_l]$: Ce sont les mêmes lettres, ce qui n'implique **aucune opération**, on passera à l'étude de la suivante :

$$\boldsymbol{D(i,j) = D(i-1, j-1)}$$

- Sinon : On teste les 3 étapes possibles dont le coût est de 1 pour chacune :
  (J'illustre le cas $i = j = 1$ ci-dessous, soit $i_l = j_l = 0$, premières lettres N et C)
  - Diagonale : Passer de $D(i-1, j-1)$ à $D(i,j)$ correspond à passer de NICHE/CHIEN à ICHE/HIEN, c'est une **substitution** de lettre soit dans M_1 (NICHE → CICHE), soit dans M_2 (CHIEN → NHIEN)) :

$$\boldsymbol{d_1 = D(i-1, j-1)}$$

  - Vers la droite : Passer de $D(i, j-1)$ à $D(i,j)$ correspond à passer de NICHE/CHIEN à NICHE/HIEN, ce qui représente soit un **ajout** dans M_1 (NICHE → CNICHE), soit une suppression dans M_2 (CHIEN → HIEN) :

$$\boldsymbol{d_2 = D(i, j-1)}$$

  - Vers le bas : Passer de $D(i-1, j)$ à $D(i,j)$ (verticalement) correspond à passer de NICHE/CHIEN à ICHE/CHIEN, ce qui représente soit à une **suppression** dans M_1 (NICHE → ICHE), soit un ajout dans M_2 (CHIEN → NCHIEN) :

$$\boldsymbol{d_3 = D(i-1, j)}$$

  On prend alors le minimum des trois résultats puisque l'on cherche la distance minimum entre les mots, auquel on ajoute le cout de la transformation (1) :

$$\boldsymbol{D(i,j) = 1 + min(d_1, d_2, d_3)}$$

On reprend la table précédente :

	j	0	1	2	3	4	5
i			C	H	I	E	N
0		0	1	2	3	4	5
1	N	1					
2	I	2					
3	C	3					
4	H	4					
5	E	5					

Question 1: Procéder à son remplissage à la main et tracer les chemins les plus courts de transformations obtenus

Vous remarquerez que l'on obtient, dans chaque case, le nombre minimum de modifications à effectuer pour passer du mot constitué des lettres de M_1 jusqu'à i au mot constitué des lettres de M_2 jusqu'à j et inversement. Vous pourrez donc vérifier votre fonction lev_rec_mem sur plusieurs exemples.

Programmation de l'algorithme itératif

On souhaite créer un dictionnaire dont les clés sont les couples (i, j) et les valeurs sont les valeurs de la table proposée dans l'algorithme itératif.

Question 2: Proposer la fonction lev(M1,M2) renvoyant le dictionnaire associé à la détermination de la distance de Levenshtein entre M1 et M2 par la méthode itérative

Vérifier :

```
>>> dico = lev('denis','demie')

>>> dico[(5,5)]
2

>>> dico = lev('niche','chien')

>>> dico[(5,5)]
4
```

Question 3: Estimer la complexité de la procédure itérative

Question 4: Créer une fonction matrice(f,M1,M2) renvoyant un array représentant le tableau associé au dictionnaire créé par la fonction f(M1,M2). On définira les valeurs non évaluées par l'algorithme à l'infini (np.inf)

Vous vérifierez que vous obtenez :

```
[[0. 1. 2. 3. 4. 5.]
 [1. 1. 2. 3. 4. 4.]
 [2. 2. 2. 2. 3. 4.]
 [3. 2. 3. 3. 3. 4.]
 [4. 3. 2. 3. 4. 4.]
 [5. 4. 3. 3. 3. 4.]]
```

Programmation de l'algorithme récursif

Question 5: Propose une fonction récursive lev_rec_ij(i,j,m1,m2) renvoyant la distance de Levenshtein entre m1 et m2 à l'aide de la méthode récursive en faisant évoluer les indices i et j

Question 6: Estimer la complexité de lev_rec_ij

Question 7: En déduire une fonction lev_rec_mem(m1,m2) récursive avec mémoïsation renvoyant le même dictionnaire que lev, à un détail près à expliquer

Vérifier :

```
>>> matrice(lev,'niche','chien')
array([[0., 1., 2., 3., 4., 5.],
       [1., 1., 2., 3., 4., 4.],
       [2., 2., 2., 2., 3., 4.],
       [3., 2., 3., 3., 3., 4.],
       [4., 3., 2., 3., 4., 4.],
       [5., 4., 3., 3., 3., 4.]])

>>> matrice(lev_rec_mem,'niche','chien')
array([[ 0.,  1.,  2.,  3.,  4., inf],
       [ 1.,  1.,  2.,  3.,  4.,  4.],
       [ 2.,  2.,  2.,  2.,  3.,  4.],
       [inf,  2.,  3.,  3.,  3.,  4.],
       [inf, inf,  2.,  3.,  4.,  4.],
       [inf, inf, inf, inf,  3.,  4.]])
```

Profitez-en pour visualiser la matrice dans le cas de deux mots identiques, par exemple : « science », ou lorsque le début ou la fin du mot présentent plusieurs lettres identiques. C'est là tout l'intérêt de la version récursive mémoïsée qui ne calcule pas tous les cas.

Applications

Téléchargez le ***TP5.2- Dictionnaire.txt*** ou sont supprimées les majuscules.

Question 8: Créer la fonction creation_Lm() renvoyant la liste des mots du dictionnaire proposé et créer la liste Lm

Dans la suite, la liste Lm sera utilisée comme variable globale.

Question 9: Créer la fonction proches(M,k) renvoyant la liste de tous les mots du dictionnaire source à une distance inférieure ou égale à k
Question 10: Ecrire les lignes de code affichant tous les mots à une distance de 1 de votre prénom
Question 11: Créer une fonction corrige(M) proposant la liste de mots à la même distance k la plus faible possible pour laquelle des mots existent dans LM
Question 12: Ecrire les lignes de code permettant de trouver l'orthographe exacte de « hydrolique »

Ne faites la suite que si vous avez de l'avance

Etapes intermédiaires

En reprenant le tableau de la question précédente, on se rend compte que pour remonter la transformation, il faut partir de la case $D(N_1, N_2)$ et effectuer les « mouvements » qui mènent vers les minimum des cases de provenance (gauche et/ou haut) pour identifier les opérations effectuées. Il existe plusieurs chemins possibles pour la distance de Levenshtein minimum obtenue.

On suppose maintenant que l'on transforme le mot M_1 (en vertical) en le mot M_2 (en horizontal), ce qui va imposer le sens de lecture de la remontée des transformation. Remarquons que lors de la réalisation de l'algorithme de Levenshtein ascendant, les déplacements signifiaient :

- Droite : Ajout dans M_1
- Gauche : Suppression dans M_1
- Diagonale : Substitution dans M_1

Mais attention, en remontant le tableau, ce qui était une suppression devient un ajout, et inversement.

On va donc procéder de la sorte, avec $i_l = i - 1$ et $j_l = j - 1$ (lettres en face de la case (i, j)) :

- Si $M_1[i_l] = M_2[j_l]$: Rien n'a été changé
  Attention : ne pas tester $D(i, j) = D(i - 1, j - 1)$ qui peut arriver par hasard
- Sinon :
  - $D(i, j) = D(i - 1, j - 1) + 1$: C'était une substitution, il faut rétablir la lettre $M[j - 1]$ avec la lettre de M_1 : $M[j_l] = M_1[i_l]$
  - $D(i, j) = D(i - 1, j) + 1$: Aller vers le bas $(i += 1)$ signifiait une suppression dans M_1, qui devient donc un ajout quand on va vers le haut : Insertion de $M_1[i_l]$ à l'indice $\max(0, \mathrm{j_l})$ dans M
    Remarque : on peut ajouter une lettre à gauche d'un mot en décalant sa première lettre vers la droite (ajout en position 0), ou en ajoutant une lettre à gauche du mot (indice -1 impossible en Python sur des str)
  - ~~$D(i, j) = D(i, j - 1) + 1$~~ : Aller à droite $(j += 1)$ signifiait un ajout dans M_1, qui devient donc une suppression quand on va vers la gauche : ~~$M[j_l]$~~

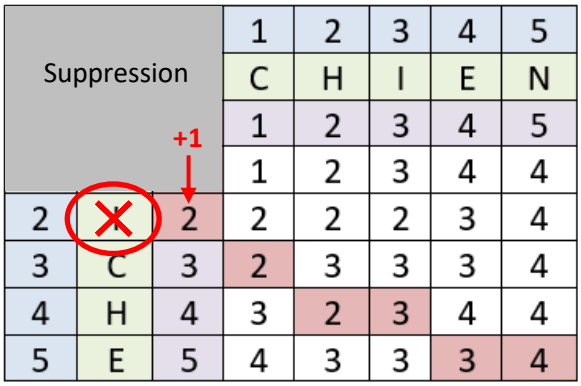
Attention : selon l'ordre de réalisation des tests, on suit une branche ou une autre, nous nous limiterons à une solution parmi celles qui peuvent exister.

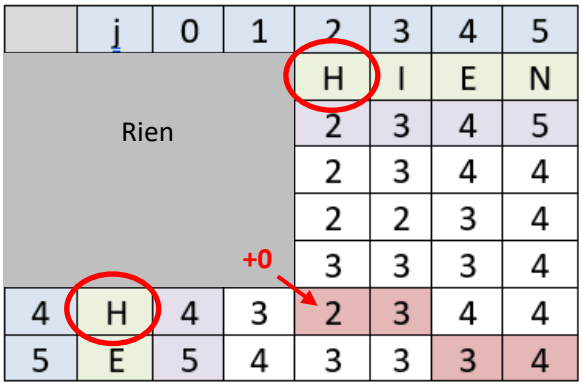
Illustrons cette procédure sur l'exemple précèdent pour lequel vous devriez avoir trouvé le tableau ci-contre. On a affiché le chemin remonté en respectant l'ordre des tests de la procédure décrite à la page précédente.

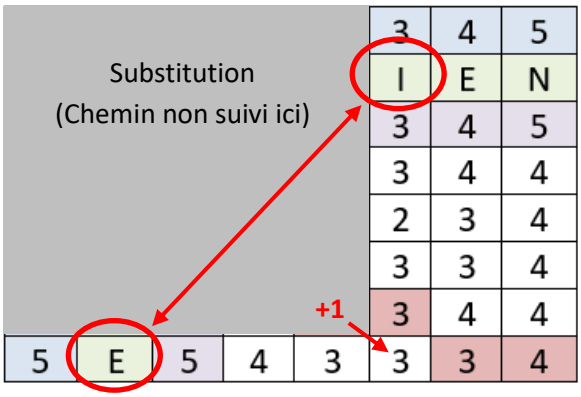
	j	0	1	2	3	4	5
i			C	H	I	E	N
0		0	1	2	3	4	5
1	N	1	1	2	3	4	4
2	I	2	2	2	2	3	4
3	C	3	2	3	3	3	4
4	H	4	3	2	3	4	4
5	E	5	4	3	3	3	4

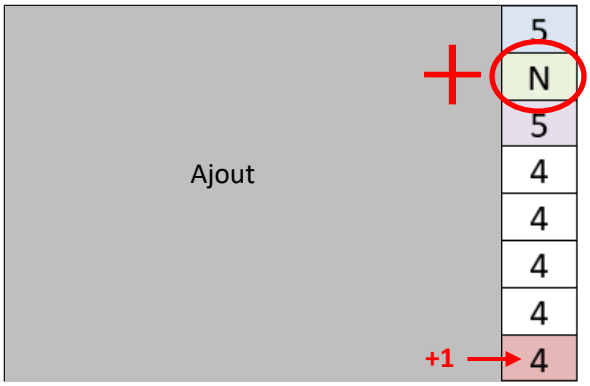
Origine	Cible	Transformation	Mot après							Indice j_l
$(5,5)$	$(5,4)$	Suppression de N : $M[4]$			C	H	I	E	N	4
$(5,4)$	$(4,3)$	Aucune			C	H	I	E		3
$(4,3)$	$(4,2)$	Suppression de I : $M[2]$			C	H	E			2
$(4,2)$	$(3,1)$	Aucune			C	H	E			1
$(3,1)$	$(2,0)$	Aucune			C	H	E			0
$(2,0)$	$(1,0)$	Ajout en $M[0]$ de I		I	C	H	E			0
$(1,0)$	$(0,0)$	Ajout en $M[0]$ de N	N	I	C	H	E			0

Pour s'aider à comprendre les étapes, il est intéressant de cacher une partie des cases et d'observer à quoi correspondent les mouvements réalisés :









On donne le code suivant, en lien ici :

```
def transformation(M1,M2):
    dico = lev(M1,M2)
    i,j,M,LM = len(M1),len(M2),M2,[M2]
    while i>0 or j>0:
        il,jl = i-1,j-1
        l1,l2 = M1[il],M2[jl]
        if i==0:
            M = #####
            j -= 1
            LM.append(M)
        elif j==0:
            M = #####
            i -= 1
            LM.append(M)
        else:
            if l1 == l2 :
                i,j = i-1,j-1
            else:
                if dico[(i,j)] == dico[(i-1,j-1)] + 1:
                    M = #####
                    i,j = i-1,j-1
                elif dico[(i,j)] == dico[(i-1,j)] + 1:
                    M = #####
                    i -= 1
                elif dico[(i,j)] == dico[(i,j-1)] + 1:
                    M = #####
                    j -= 1
                LM.append(M)
    LM.reverse()
    return LM
```

Question 13: Créer la fonction sub(M,i,t) substituant par t la lettre d'indice i de M (str)

Question 14: Créer la fonction ins(M,i,t) insérant la lettre t à l'indice i de M (str)

Question 15: Créer la fonction sup(M,i) supprimant la lettre à l'indice i de M (str)

Vérifier :

```
>>> sub('essai',0,'1')
'1ssai'

>>> ins('essai',0,'1')
'1essai'

>>> sup('essai',0)
'ssai'
```

Question 16: Compléter la fonction transformation(M1,M2) qui renvoie la liste des étapes pour passer de M1 à M2 avec l'algorithme de Levenshtein

Question 17: Utiliser votre fonction afin d'afficher les étapes du passage entre deux chaines de caractères

Exemple si vous respectez l'ordre des test ci-dessus :

```
>>> transformation('niche','chien')
['niche', 'iche', 'che', 'chie', 'chien']
```

Une autre approche

Préambule : Jusqu'à ce que je trouve qui a déjà proposé ce qui suit, cette méthode s'appelle « Defauchy (2023) » 😊.

On reprend les notations introduites précédemment :

- M_1^i est le mot M_1 privé de les i premières lettres
- M_2^j est le mot M_2 privé de ses j premières lettres

On souhaite mettre en place un dictionnaire représentant par ses clés (i,j) une tableau à deux dimensions contenant les valeurs $D(i,j)$ obtenues lors de l'exécution de l'algorithme récursif mémoïsé de la première partie de ce TP (cf. graphe) avec (i,j) les entiers associés aux mots restant à traiter M_1^i et M_2^j. **C'est de là que cette méthode est venue, en voulant coller au graphe proposé en début de sujet**.

Question 18: En vous basant sur la fonction lev_rec_mem, proposer une fonction lev_rec_mem_bis(M1,M2) renvoyant le dictionnaire souhaité

Vérifier :

```
>>> matrice(lev_rec_mem,'oui','non')
array([[0., 1., 2., 3.],
       [1., 1., 1., 2.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.]])
```

Chaque terme $D(i,j)$ de cette matrice représente le nombre d'étapes minimum restantes pour aller d'un des mots M_1^i/M_2^j à l'autre. Par exemple :

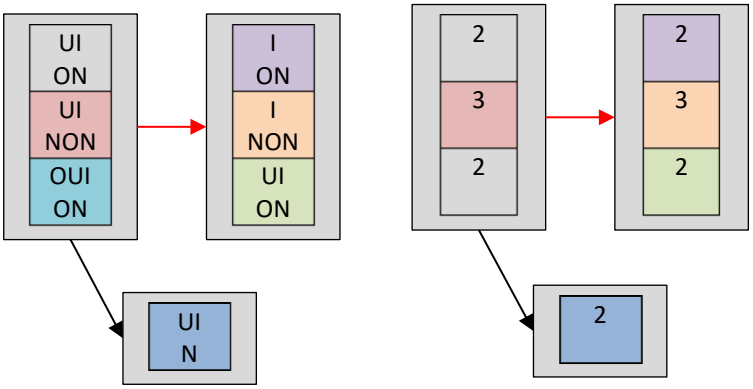
	j	0	1	2	3
i		N	O	N	
0	O	3	2	∞	∞
1	U	3	2	2	2
2	I	3	2	1	1
3		3	2	1	0

La case $D(1,2) = 2$ en rouge indique que, peu importe le chemin pour arriver à cette case, il reste au minimum 2 étapes entre OUI privé de sa première 1° lettre $(i = 1)$ « UI » et NON privé de ses 2 premières lettres $(j = 2)$ « N »

	j	0	1	2	3
i		N	O	N	
0	O	3	2	∞	∞
1	U	3	2	2	2
2	I	3	2	1	1
3		3	2	1	0

La case $D(3,0) = 3$ en rouge indique que, peu importe le chemin pour arriver à cette case, il reste au minimum 3 étapes entre OUI privé de ses 3 premières lettres $(i = 3)$ « » et NON privé de 0 premières lettres $(j = 0)$ « NON »

En reprenant un extrait du graphe du début de ce sujet, à partir d'une situation quelconque, il y a 3 possibilités :



	j	0	1	2	3
i		N	O	N	
0	O	3	2	∞	∞
1	U	3	2	2	2
2	I	3	2	1	1
3		3	2	1	0

Remarquez comment remplir ce tableau, un peu comme pour l'algorithme ascendant, mais en partant d'une situation différente. En partant d'en bas à droite, il faut remplir le tableau en prenant :

- La valeur dans le tableau de la case en dessous à droite si les lettres sont identiques
- Le minimum des 3 valeurs (quand elles existent) en dessous et/ou à droite, plus 1

	j	0	1	2	3
i		N	O	N	
0	O				
1	U				
2	I				
3					0

On ne trouvera pas les valeurs infinies du premier coup, elles sont liées à la présence de lettres identiques permettant un seul appel en diagonal au lieu de 3 dans les 3 directions. Mais il est inutile d'y réfléchir, on remontra les transformations sans y passer. C'est là un avantage de la réalisation du dictionnaire par la méthode récursive mémoïsée plutôt que par une méthode ascendante.

Question 19: Proposer une fonction lev_bis(M1,M2) réalisant cette table par la méthode ascendante proposée et vérifier les matrices obtenues

Pour trouver les transformations de OUI à NON ou inversement, à partir de la case $D(0,0) = 3$ correspondant à la distance minimale renvoyée par l'algorithme, il faut remonter le chemin des minimums pour arriver à 0. On sait que tout déplacement coûte 1 sauf si les lettres sont identiques, auquel cas le chemin en diagonale est obligatoire.
Voici les 3 chemins les plus courts entre OUI et NON :

	j	0	1	2	3
i		N	O	N	
0	O	3	2	∞	∞
1	U	3	2	2	2
2	I	3	2	1	1
3		3	2	1	0

	j	0	1	2	3
i		N	O	N	
0	O	3	2	∞	∞
1	U	3	2	2	2
2	I	3	2	1	1
3		3	2	1	0

	j	0	1	2	3
i		N	O	N	
0	O	3	2	∞	∞
1	U	3	2	2	2
2	I	3	2	1	1
3		3	2	1	0

Remarques :

- Le passage de $(0,1)$ à $(1,2)$ est obligatoire puisque les lettres sont identiques (O) et ne change pas le 2
- On retrouve les 3 chemins obtenus en début de sujet

Voici l'exemple du pire chemin réalisable :

- De $(0,0)$ à $(1,0)$: $C = 1$, il reste toujours au minimum 3 étapes à réaliser entre UI et NON
- De $(1,0)$ à $(2,0)$: $C = 2$, il reste toujours au minimum 3 étapes à réaliser entre I et NON
- De $(2,0)$ à $(3,0)$: $C = 3$, il reste toujours au minimum 3 étapes à réaliser entre _ et NON
- Il reste alors les 3 étapes horizontales pour un coût de 3, soit en définitive, $C = 6$

	j	0	1	2	3
i		N	O	N	
0	O	3	2	∞	∞
1	U	3	2	2	2
2	I	3	2	1	1
3		3	2	1	0

On voit que les déplacements représentent les opérations suivantes :

- Aller en diagonale correspond à une substitution dans M_1 ou dans M_2
- Aller vers le bas correspond à un retrait dans M_1 ou un ajout dans M_2
- Aller vers la droite correspond à un retrait dans M_2 ou un ajout dans M_1

Fixons le départ M_1 pour aller à M_2. On a donc :

- Diagonale : substitution dans M_1
- Bas : retrait dans M_1
- Droite : ajout dans M_1

Par ailleurs, contrairement à la méthode précédente, on va reconstruire le mot dans le bon ordre.

On donne le code suivant, en lien ici :

```
def transformation_bis(M1,M2):
    dico = lev_rec_mem_bis(M1,M2)
    i,j,M,LM = 0,0,M1,[M1]
    il = i
    while i<len(M1) or j<len(M2):
        if i==len(M1):
            l2 = M2[j]
            M = #####
            j += 1
            il += 1
            LM.append(M)
        elif j==len(M2):
            M = #####
            i += 1
            LM.append(M)
        else:
            l1,l2 = M1[i],M2[j]
            if l1 == l2 :
                i,j = i+1,j+1
                il += 1
            else:
                if dico[(i,j)] == dico[(i+1,j+1)] + 1:
                    M = #####
                    i,j = i+1,j+1
                    il += 1
                elif dico[(i,j)] == dico[(i+1,j)] + 1:
                    M = #####
                    i += 1
                elif dico[(i,j)] == dico[(i,j+1)] + 1:
                    M = #####
                    j += 1
                    il += 1
                LM.append(M)
    return LM
```

Question 20: Compléter la fonction transformation_bis afin qu'elle renvoie une transformation de M1 en M2 et vérifier qu'elle fonctionne

Remarque intéressante : Faisons la somme des tables obtenues avec les deux méthodes abordées dans ce TD :

	j	0	1	2	3
i		N	O	N	
0	O	0	1	2	3
1	U	1	1	1	2
2	I	2	2	2	2
3		3	3	3	3

+

	j	0	1	2	3
i		N	O	N	
0	O	3	2	3	3
1	U	3	2	2	2
2	I	3	2	1	1
3		3	2	1	0

=

	j	0	1	2	3
i		N	O	N	
0	O	3	3	5	6
1	U	4	3	3	4
2	I	5	4	3	3
3		6	5	4	3

On fait apparaître les longueurs minimales des chemins pour les transformations passant par les cases concernées et les chemins optimaux en suivant les 3 (déplacement en diagonale si lettres identiques).