Lycee Charlemagne | MPSI2 | Annee 2021/22

I.T.C.

Février

L'objectif de ce T.D. est (après un peu d'entrainement sur un fichier contenant un dictionnaire francophone) de réaliser un correcteur orthographique.

Objectif informatique :
- ouverture et lecture de fichiers issus du disque dur
- manipulations de listes
- recherche d'éléments réalisant un maximum
- manipulation de dictionnaires (au sens informatique du terme)
- distance de LEVENSHTEIN (théorie des graphes)
- indice de JACQUARD

Un fichier, placé dans le répertoire actif contient une quantité importante de mots du dictionnaire français (suffisant pour tout ce qui va être fait ici. Il s'appelle FRENCH.TXT (il est donc de type « texte brut »).
On va l'ouvrir en mode lecture, et se livrer à quelques jeux OuLiPiens.

VOCALISES

ACCENTUATION, AUTOCUISEUR, SOUVERAIN, SAOUDITE, SAPEUR-POMPIER, APOCALYPTIQUE, AÉRODYNAMIQUE, YOUGOSLAVIE et TYPOGRAPHIQUE

Testez le programme suivant :

```
import os
Dico = open('FRENCH.TXT', 'r')
Scooby = []
for mot in Dico :
....if 'a' in Mot and 'e' in Mot and 'i' in Mot and 'o' in Mot and 'u' in Mot :
........Scooby.append(Mot)
print(Scooby)
```

A quoi sert la ligne import os ? Et pourquoi pas from os import *, ou import os as oss117 ?

.
.

Sans cette ligne import os, que se passe-t-il ?

.
.

Combien de mots trouvez vous ?

.
.

Et si vous voulez plusieurs lettres a ?

.
.

Et si vous voulez une seule fois chaque voyelle ?

.
.
.

Au fait, et le y là dedans ?

.
.

EXCES

INDIVISIBILITÉ, CORROBORATION, ANTICOMMUNISME, BAOBAB, ABRACADABRANT, TARTELETTE, CONCUPISCENCE

```
import os
Fichier = open('FRENCH.TXT','r')
Scooby, Record = ' ', 0
for Mot in Fichier :
....Compt = Mot.count('i')
....if Compt > Record :
........Scooby = Mot
........Recrod = Compt
print('record ', Record, Scooby)
```

Quelles lettres ont été testées dans les exemples ?

.
.

Et si vous n'avez pas droit à la fonction count ?

.
.

Pourquoi avez vous recopié `Recrod = Compt` ?

.
.

Et si vous cherchez les e, mais aussi les é, les è, les ê ?

.
.
.
.

Mais si plusieurs mots réalisent le record, que faut il faire ?

.
.
.
.
.

Et comment trouver le mot qui réalise le record quand on tente toutes les lettres de l'alphabet (le record des records, c'est pour a, e, t ?) ?

.
.

MOT CACHE

RESTRUCTURATION, SOUSCRIPTEUR, SUPERCHERIE, SOUSCRIPTEURSURENCHÉRIE

Le malin m'a inspiré le programme suivant :

```
import os
Dico=open('FRENCH.TXT','r')
for mot in Dico :
....if Mystere(mot,'satan') :
........print(mot)
print('Fini')
Dico.close()
```

```
def Mystere(Mot1,Mot2) :
....compt = 0
........for lettre in Mot1 :
............if lettre == Mot2[compt] :
................compt+=1
................if compt==len(Mot2) :
....................return(True)
....return(compt == len(Mot2))
```

A quoi sert ma ligne `print('Fini')` ?

.
.

A quoi sert ma ligne `Dico.close()` ?

.
.

Pouvez vous me dire si on verra à l'affichage les mots SOUS-TRAITANCE, SATURATION, SPECIALISATION, CONSTATATION, TRANSPIRATION, DÉMOCRATISATION.

.
.

Ecrivez les commentaires qui manquent dans le programme. Décrivez[1] l'évolution des variables lettre, compt,

1. (par un tableau, évidemment, on est en sciences, pas en dissertation d'histoire-géo, et encore, même en histoire géo, c'est tellement plus

`lettre == Mot2[compt]` quand s'exécute la boucle `for lettre in Mot1` de la procédure `Mystere(mot, 'satan')` pour l'un des six `mot`s ci-dessus *(de soustraitance à démocratisation)*.

.
.
.
.
.
.

Expliquez le rôle des lignes

```
if compt==len(Mot2) :
....return(True)
```

.
.
.
.

J'ai remplacé SATAN par un prénom de la classe et trouvé 54 mots dont AMERICAIN, MÉDICAMENT, DOMINICAL, MUNICIPALES... c'était qui ?
Et pour MARTIN-PECHEUR ?
Et pour MATHÉMATIQUES, SYMPATHIQUES, CINÉMATOGRAPHIQUES ?
Et pour ANTICONSTITUTIONNEL ?

.
.

Et pour TRANSFORMATION, RECOMMANDATION ?
Et pour AQUARELLE, NATURELLE ?
Et pour PANORAMA, ENTHOUSIASME ?
J'ai enelvé un des mots ['archéologie', 'archéologique', 'archéologue', 'archéologues', 'biotechnologie', 'biotechnologies', 'chaloupe', 'charlotte', 'chlore', 'chronologie', 'chronologique', 'psychologie', 'psychologique', 'psychologiquement', 'psychologiques', 'psychologues', 'rééchelonnement', 'tchécoslovaque', 'tchécoslovaques', 'Tchécoslovaquie', 'technologie', 'technologies', 'technologique', 'technologiques', 'technologiste', 'technologue], lequel ?
Qui a le record avec plus de 300 mots : Dan, Omi, Lae, Côme [2] ?

.
.

Et pour CHAMAILLE ?
Et qui pour ASPERGE, SERINGUE, SUBMERGER, SYNERGIE ?
Et quipour INVOLONTAIRE et MILLIONAIRE ?
Pour SOLÈNE [3], un seul mot réagit positivement pour LOUISE, et une floppée pour son autre fille ANAÏS *(si on ne met pas le tréma)*.
Allez, pour LOUISE, c'est quoi ?

.
.

Et qui est le mot caché du titre ?

..

Les trois exercices suivants ne sont pas obligatoires.
Vous pouvez, si vous préférez, passer tout de suite au correcteur orthographique.

ULCERATIONS
PSYCHODRAME TRANSLUCIDE PROBLÉMATIQUES

Ecrivez un programme qui cherche le mot le plus long du dictionnaire, mais dont toutes les lettres sont distinctes (ce n'est donc pas "ANTICONSTITUTIONNELLEMENT", ni même un quelconque "DIMÉTHYLPHÉNOL…ETC").

L'Oulipo de *Queneau*, *LeLyonnais*, *Perec* et les autres avait "ULCÉRATIONS". "PSYCHODRAME TRANSLUCIDE SYMPATHIQUE" ne m'ont pas l'air mal. Et si les lettres accentuées sont différentes entre elles, j'ai "CINÉMATHÈQUES DÉMOGRAPHIQUES PROBLÉMATIQUES".

Ecrivez un programme qui cherche le mot du dictionnaire qui utilise le plus de lettres différentes *("PORTEZ CE VIEUX*

clair avec des tableaux et de l'infographie)

2. avec accent, j'ai juste CHÔMAGE et ses variantes
3. insolence, somnolence

WHISKY AU JUGE BLOND QUI FUME" *est une phrase, pas un mot, mais* "PSYCHOLOGIQUEMENT" *est pas mal)*.

CRESCENDO DECRESCENDO

ABEILLE, BOUSSOLE, MORSURE, CEINTURON

Toujours notre fichier `FRENCH.TXT`. On doit trouver cette fois les mots comme ABEILLE, CEINTURON, IMPOLI, BOUSSOLE, MOUSSE et MORSURE.
Qu'ont ils de particulier ? Les lettres y sont dans l'ordre alphabétique croissant, puis décroissant. Ou décroissant puis croissant (CAEN), ou même seulement croissant (BOSS) ou seulement décroissant (VOLIGE). $a^{b^{e^{i^{ll_e}}}}$, $b^{o^{u^{ss^{o_{l_e}}}}}$, $m^{o^{r^{s^{u^{r_e}}}}}$, $s_{o_{n_{ge}}}$, $c_{a^{e^n}}$, mais pas $C^{h^{o^{q^{u^{e^t}}}}}$.
Allez, c'est à vous !

ANAGRAMMES

Un ADONIS DANOIS veut ACHETER un HECTARE pour la NICHE de son CHIEN. Un AMATEUR de MEETINGS (ou un MARTEAU dans un GISEMENT ?) Un ARBITRE va s'ABRITER pour échapper à l'AIGLE AGILE qui fait ALLIANCE avec une CANAILLE.

Ah il y en a des couples de mots anagramme l'un de l'autre.
Ouvrez le fichier `FRENCH.TXT` et cherchez les couples d'anagrammes. Vous avez le droit de convertir les chaines en listes (`list(Mot)`), de trier les listes (`sorted(L)`).
Sur le toit, avez vous une gouttière ou une girouette ?

Comment savoir rapidement si deux listes sont égales sans être forcément dans le même ordre ? On les trie toutes les deux, et on regarde si les listes triées sont égales. Il y a aussi des solutions avec les ensembles, en utilisant une différence symétrique.

On va ouvrir le fichier, le lire. Comme on va le lire deux fois, on le stocke dans une variable.
On transforme les mots en listes une bonne fois pour toutes avec `list(Mot)`.

On ne testera ensuite que les mots de même longueur.
Comme la fonction `sorted` est un peu lourde, on évitera de trop la solliciter.
Et si finalement on la sollicitait une seule fois pour chaque mot, en créant une liste des mots triés, en gardant une liste des mots véritables.

On évitera aussi de dire qu'un mot est un anagramme de lui même.
Et on évitera de tester ASPIRANT avec PARTISAN puis PARTISAN avec ASPIRANT,
on ne va donc parcourir qu'une moitié de produit cartésien :
`for k in range(len(L))`
suivi de `for i in range(k)` tout court.

Allez, c'est à vous...

Le physicien Etienne Klein est grand amateur d'anagrammes, en voici quelques uns piqués dans ses oeuvres :

1• La théorie de la relativité restreinte ->	a• vérité théatrale et loi intersidérale
2• Le Radeau de la Méduse ->	b• tissèrent l'espace et la lumière
3• Saint-Germain des Près ->	c• un lot de vestes d'Hérodote
4• Hôtel des ventes de Drouot ->	d• A la lueur fauve d'un gros lampion dépoli, et gouvernant mon émoi, je décrypte des cartouches
5• Le commandant Cousteau ->	e• démasqua le désir
6• L'origine de l'univers ->	f• tout commença dans l'eau
7• Jean-François Champollion, conservateur du département d'égyptologie au musée du Louvre ->	g• matins nègres de Paris
8• L'origine du monde, Gustave Courbet ->	h• Oui, et la poser n'est que vanité orale
9• Le marquis de Sade ->	i• un vide noir grésille
10• Etre ou ne pas être, voilà la question ->	j• au-delà de la démesure
11• Et les particules élémentaires ->	k• ce vagin où goutte l'ombre d'un désir

Passons sans tarder au correcteur orthographique.

Lycee Charlemagne | MPSI2 | Annee 2021/22

CORRECTEUR ORTHOGRAPHIQUE

Le but de ce problème est de mettre au point un correcteur orthographique pour traitement de texte. Vous savez, cette chose sous OPEN-OFFICE [4] qui, quand vous tapez un mot qui n'existe pas *(faute de frappe, pas eu assez de dictées au collège, nom propre...)* vous propose des mots du dictionnaire qui sont corrects et proches du mot que vous avez tapé. L'ordinateur dispose donc d'un lexique *(notre `FRENCH.TXT` des DS et IS)* dans lequel sont stockés des mots correctement orthographiés.
Ecrivez un script utilisant le module `os` qui lit le fichier `FRENCH.TXT` crée la liste `Dico` des mots du lexique [5].

Transformation de FRENCH.TXT en Dico.

Lycee Charlemagne | MPSI2 | Annee 2021/22

Distance de Levenshtein

On va définir la distance entre deux mots dite de LEVENSHTEIN. Quand les deux mots sont égaux, la distance est nulle, et plus elle est grande, moins les mots se ressemblent.

Pour passer d'un mot `MotA` à un mot `MotB`, on a droit aux étapes suivantes :

- E_i : effacement de la lettre d'indice `i` de `MotA`
- $I_{i,j}$: insertion en poisition `i` dans `MotA` de la lettre d'indice `j` de `MotB`
- $R_{i,j}$: remplacement de la lettre d'indice `i` de `MotA` par la lettre d'indice `j` de `MotB`.

La distance de LEVENSHTEIN est le plus petit nombre d'étapes pour passer de `MotA` à `MotB`.
Par exemple la distance de `nicolas` à `solene` est de 6 :

`nicolas`	$R_{0,0}$	`sicolas`	E_1	`scolas`	E_1	`solas`	$R_{3,3}$	`soles`	$R_{4,4}$	`solen`	$I_{5,5}$	`solene`

Calculez la distance de `solene` à `nicolas` (en justifiant) [6].

Distance (et étapes) de SOLENE à NICOLAS.

Justifiez que la distance de `sucri` à `christ` est de 5.

4. ne me dites pas que vous vous êtes laissé racketer par MICROSOFT avec WORD !
5. dans tout ce qui suit, on parlera de lexique, plutôt que de dictionnaire, réservant ce mot aux objets `Python` appelés dictionnaires
6. pardon ? je vous prends pour des idiots ?

Distance (et étapes) de SUCRI à CHRIST.

Justifiez que la distance de `chien` à `niche` est de 4.

Distance (et étapes) de CHIEN à NICHE.

L'algorithme est le suivant[7] :

```
def Leven(MotA, MotB) : #distance de Leveshtein
....LA, LB = len(MotA), len(MotB)
....D = [[0 for j in range(LB+1)] for i in range(LA+1)]
....for i in range(LA+1) :
........D[i][0]=i
....for j in range(LB+1) :
........D[0][j]=j
....for i in range(1, LA+1) :
........for j in range(1, LB+1) :
............if MotA[i-1] == MotB[j-1] :
................cout = 0
............else :
................cout = 1
............D[i][j]=min(D[i-1][j]+1, D[i][j-1]+1, D[i-1][j-1]+cout)
........#print('Etape ', i)
........#for k in range(LA+1) :
........#....print(D[k])
....return(D[LA][LB])
```

On ne demande pas de justifier que ce script calcule effectivement la distance *(ce serait du domaine de l'option informatique et pas de l'I.P.T.)*.

Pouvez vous compacter en une ligne le bloc

```
if MotA[i-1] == MotB[j-1] :
...cout = 0
else :
...cout = 1
```

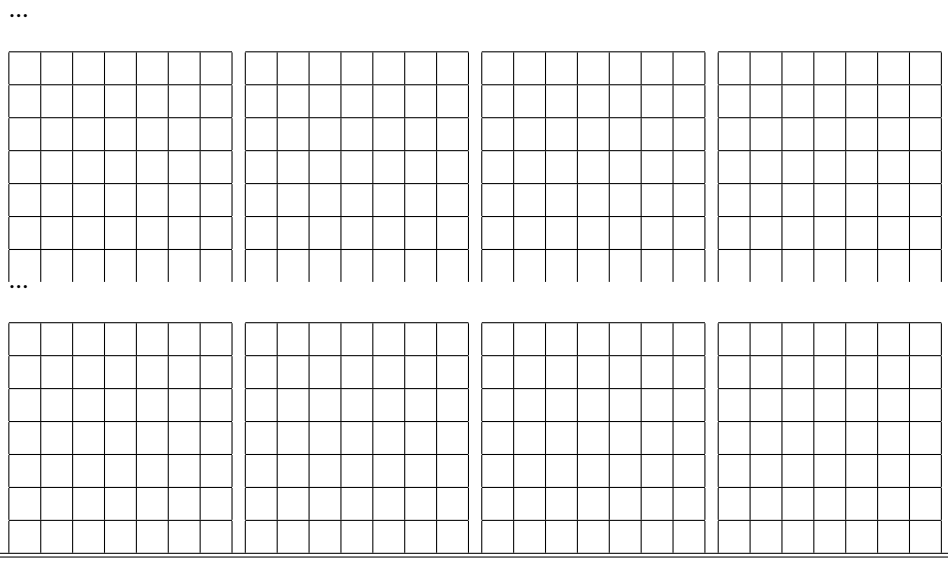
Compactage du bloc en une ligne.

Indiquez le résultat affiché si vous exécutez `Leven('sucri', 'christ')` après avoir enlevé les dièses du bloc

7. on ne justifiera pas qu'il calcule effectivement la distance, il faudrait un prochain TD

```
print('Etape ', i)
for k in range(LA+1) :
....print(D[k])
```

Exécution de Leven('SUCRI', 'CHRIST').



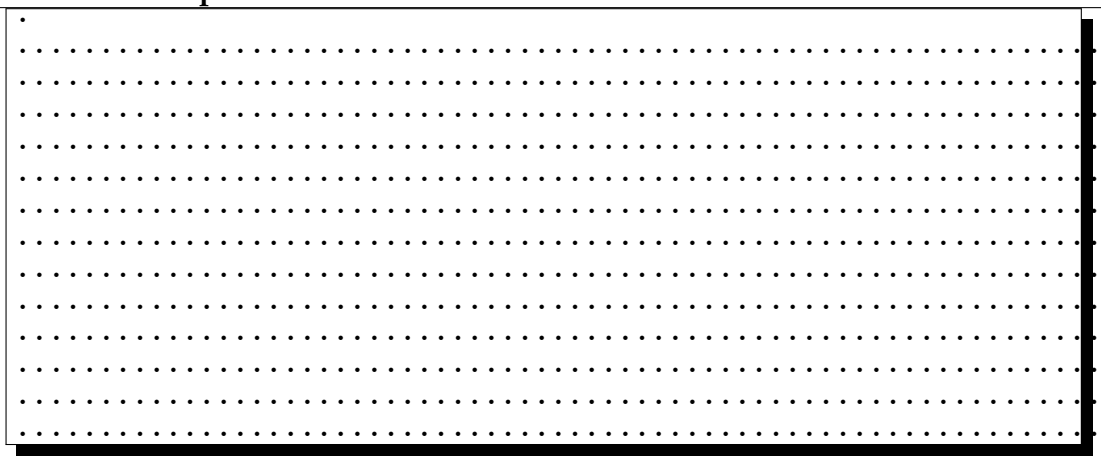
Ecrivez un script qui prend en entrée un mot `Mot` et répond un quadruplet fait d'un booléen, d'un entier, d'une liste et d'un flotant :

• si `Mot` est dans `Dico`, la réponse est `[True, 0, [Mot], t]` où `t` est le temps passé pour la recherche

• si `Mot` n'est pas dans `Dico`, la réponse est `[False, d, [...], t]`, `d` étant la plus courte distance de LEVENSHTEIN de `Mot` aux mots du lexique, la liste contenant ensuite les mots du lexique à distance `d` de `Mot`, et `t` est le temps consacré à la recherche [8].

Par exemple, il devra répondre `(False, 2, ['saline', 'silence', 'silène', 'soient', 'solde', 'soldée', 'soleil', 'solennel', 'solide', 'solive', 'sonne'], 3.38)` si on lui a proposé `solene`, et `[False, 2, ['citron', 'patron', 'piéton', 'pigeon', 'pignon', 'pilon', 'pion', 'thon'], 3.39]` si on lui donne le mot erroné `pithon` *(sachant que j'ai un dictionnaire qui ne connaît pas le mot* `python`*)* [9]

Recherche des mots proches.



Sachant que l'exécution d'un tel programme prend de 3 à 7 secondes suivant la longueur de `Mot`, on va devoir optimiser si on veut pouvoir parcourir rapidement un texte complet et proposer des corrections d'ailleurs plus pertinentes que celles obtenues ici.

8. le module time contient la fonction `clock()` qui lit l'heure
9. et j'ai eu `[False, 1, ['coquille', 'fouille', 'houille', 'rouille']], 3.91]`, mais je ne sais plus pour quel mot

Trigrammes

On va travailler sur des triplets de lettres *(appelés trigrammes)*, découper les mots du lexique et le mot à analyser en trigrammes.
Pour découper un mot en trigrammes : on colle un `$` en début et en fin de mot, puis on découpe tous les paquets de trois lettres successives *(combien un mot de n lettres a-t-il de trigrammes ?)*.
Par exemple : `nicolas` donne `$nicolas$` puis `['$ni', 'nic', 'ico', 'col', 'ola', 'las', 'as$']`.
Ecrivez une procédure `Trigrammes` qui prend en entrée un mot `Mot` et crée la liste de ses trigrammes.

Procédure Trigrammes.

```
def Trigrammes(Mot) :
.....
.....
.....
.....
.....
.....
.....
.....
```

Et on va utiliser utiliser des dictionnaires `Python`.

Qu'est ce qu'un {dictionnaire} `Python` ? Un objet un peu plus complexe que les 'chaînes de caractères' et des [listes].
Un dictionnaire est proche d'une liste (ses éléments sont modifiables), mais les éléments ne sont pas indexés en séquence (l'appel de `L[i]` avec `i` dans `range(len(L))` ; les éléments enregistrés ne sont pas dans un ordre immuable. L'accès aux éléments se fait par une clef (en anglais KEY), qui sera ici alphabétique.

Par exemple, un dictionnaire nomme `MPSI` pourra être constitué des noms des enseignants de la classe avec pour clef la matière enseignée :
`MPSI2 = {'Maths' : ['Choquet', 'Wattiez'], 'Physique' : ['Chevalier', 'Chabane'] : 'SII', ['Papanicola'] : 'Philo' : ['Seguret', 'Ayoun']}`

On veut savoir qui ensigne la philo : `print(MPSI2['Philo'])` donne `['Seguret', 'Ayoun']`.

On a oublié l'anglais ? Facile : `MPSI['Anglais']=['Jeanbourquin']` *(pas besoin de préciser qu'on allonge par* append*)*.
Et voilà le nouveau `MPSI` : `{'Maths' : ['Choquet', 'Wattiez'], 'Physique' : ['Chevalier', 'Chabane'], 'SII' : ['Papanicola'], 'Anglais' : ['Jeanbourquin'], 'Philo' : ['Seguret', 'Ayoun']}`.

On a oublié que les MPSI1 avaient un autre professeur : `MPSI['SII'].append('DeBernardi')`.

On veut voir les clefs : `print(MPSI.keys())` donne `['Maths', 'Anglais', 'Physique', 'SII', 'Philo']` *(l'ordre étant imprévisible)*.
1- Que donne `MPSI['Maths'].remove('Choquet')` ?
2- Quelle instruction devez vous taper pour ajouter l'informatique et l'I.T.C..
3- Quelle instruction devez vous taper pour que madame **CHEVALIER** s'appelle **CHEVALIER-THERY** ?
4- Quel est l'effet de **for A in MPSI : MPSI[A].append('Sucri')** ?
5- La `PCSI` a aussi un dictionnaire du même type. Tapez une instruction qui affecte l'I.T.C. à la réunion des professeurs de maths, de physique, de chimie et de O.Brunet, si toutefois il n'y a pas déjà un prof d'I.T.C. dans le dictionnaire.

Dictionnaire MPSI.

Un petit test sur les dictionnaires ? Que va donner ce script ?

```
texte ='longtemps je me suis couché de bonne heure'
table = {}
for lettre in texte :
....if lettre in table.keys() :
........table[lettre] += 1
....else :
........table[lettre] = 0
print(table)
```

Citation de Proust.

Votre mission est maintenant de parcourir le lexique `Dico` et de créer le dictionnaire `BonsTrigrammes` des trigrammes. Les clefs seront les trigrammes. Chaque trigramme pointe vers la liste des mots de `Dico` contenant ce trigramme.

Par exemple, `BonsTrigrammes['nic']` *répond ['adminicule', 'arsenic', 'canicule', 'communication', 'communicationnel', 'communications', 'corniche', 'dénichant', 'dénicher', 'dominical', 'fornicateur', 'mécanicien', 'mécaniciens', 'Munich', 'municipal', 'municipale', 'municipales', 'municipalité', 'municipalités', 'municipaux', 'nicher', 'nichons', 'nickel', 'péniche', 'pleurnicher', 'romanichel', 'Romanichelle', 'technicien', 'techniciens', 'technicité', 'télécommunications', 'yannick'].*

Ce programme sera exécuté à l'ouverture de logiciel de traitement de texte, une fois pour toutes.

Création du dictionnaire BONSTRIGRAMMES.

```
.
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
```


Ressemblance.

Maintenant, étant donné un mot `Mot` à corriger, on va chercher plus efficacement les mots du lexique ayant le plus de trigrammes en commun avec lui, ce qui devrait être un bon indice de ressemblance.

En utilisant `BonsTrigrammes` et `Trigrammes`, Ccréez le dictionnaire `Commun` des mots ayant des trigrammes communs avec `Mot`. Pour chacun, la clef sera le mot, et elle renverra sur le nombre de trigrammes communs avec `Mot`.

Toutefois, comme les mots longs ont plus de trigrammes et donc plus de chances d'avoir des trigrammes communs avec votre mot, il va falloir modifier légèrement le décompte. On va calculer un taux de ressemblance entre deux mots `MotA` et `MotB`, inspirée de l'indice de JACQUARD :

L'indice de Jacquard de deux ensembles A et B est $Jac(A,B) = \dfrac{Card(A \cap B)}{Card(A \cup B)}$.

Que signifie $Jac(A,B) = 0$? Que signifie $Jac(A,B) = 1$? Que signifie $Jac(A,B) = 2$?

Indice de Jacquard.

La formule ici utilisée sera $taux = \dfrac{NbTrig(MotA,\ MotB)}{len(MotA) + len(MotB) - NbTrig(MotA,\ MotB)}$ où `NbTrig(MotA, MotB)` compte le nombre de trigrammes communs entre `MotA` et `MotB`.
Afficher alors la liste des mots ayant un taux plus grand que $0,2$.

Mots d'indice plus grand que 0.2.

Justifiez qu'un taux égal à 1 correspond à "`MotA` était dans le lexique".

Cas des mots d'indice 1.

Terminez en trouvant pour un mot tapé (erroné), les mots ayant le plus fort indice de Jacquard.
Vérifiez la pertinence de votre correcteur.

Lycee Charlemagne I.T.C.

Un script du serpent satanique.

On décrypte le script :

`import os`	on importe le module
`Dico=open('FRENCH.TXT','r')`	on ouvre un document et on l'affecte à une variable
`for mot in Dico :`	on lit ligne à ligne le contenu de ce fichier [10]
`....if Mystere(mot,'satan') :`	on fait un test en sollicitant une procédure
`........print(mot)`	si le test est valide on affiche le mot
`print('Fini')`	on prévient que c'est fini
`Dico.close()`	on fait les choses proprement, bon sang !

On étudie alors la procédure, qui prend en entrée deux variables. Ce sera ici le mot lu dans le fichier texte, et un second mot, 'satan' dans notre exemple. On initialise un `compteur` à `0`, qui ne va pas se contenter de compter des égalités, mais qui va aussi faire avancer caractère à caractrère dans le mot `Mot2`.

`def Mystere(Mot1,Mot2) :`	On prend donc deux chaines ou listes en variables.
`....compt = 0`	On initialise un compteur.
`........for lettre in Mot1 :`	On parcourt une à une les lettres du mot `Mot1`.
`............if lettre == Mot2[compt] :`	On regarde si il y a occurence avec une certaine lettre du mot `Mot2`.
`................compt+=1`	Si il y a occurence, on avance le compteur. On passera donc à la comparaison à la lettre suivante de `Mot2`.
`................if compt==len(Mot2) :`	On teste si on est allé jusqu'au bout des lettres de `Mot2`.
`....................return(True)`	Si oui, on répond `True`, sinon, on continue à comparer lettre à lettre.
`....return(compt == len(Mot2))`	On renvoie un test. C'est un booléen, qui est donc évalué et vaut `True` ou `False`

Je sais, pour la fin, vous auriez tapé

```
if compt == len(Mot2) :
....return(True)
else :
....return(False)
```

ce qui est correct, mais aussi redondant que si vous écriviez `if (a compt==len(Mot2))==True.`
celà dit, je vous connais, et vous m'écrivez effectivement des `if Test(a)==True :` *quand* `Test(a)` *est lui même un booléen. Mais c'est normal, vous avez juste un bac S, pas un bac scientifique.*

Que fait ce script ? Exactement ce que vous feriez à la main avec par exemple les mots `constatation` et `satan`, qui sont deux chaînes de caractères.

Vous prenez le `s` de satan c'est `Mot2[compt]` avec `compt=0`.

Vous prenez une à une les `lettre`s de `constatation` : `'c'`, puis `'o'`, puis `'n'` *(sans malice, ça fait* `con`*)*.

Pour l'instant, aucune n'est égale à `'s'`, vous traversez le test sans rien faire.

Vous arrivez à `'s'` de `conStatation` ; là, il y a égalité entre lettre et `Mot2[0]` ; le teste est valide, compt passe à 1. Les tests se feront alors entre `lettre` et `Mot2[1]`, c'est à dire `'a'`.

La lettre suivante (`'t'`) ne donne pas un test valide, on continue à avancer.

La lettre suivante (`'a'`) donne un test valide, on passe `compt` à `2`. Il n'a pas encore atteint la valeur critique `len(Mot2)`, on continue donc.

lettre	c	o	n	s	t	a	t	a	t	i	o	n
Mot2[compt]	s	s	s	s	a	a	t	a	n	n	n	n
test	F	F	F	T	F	T	T	T	F	F	F	F
compt	0	0	0	1	1	2	3	4	4	4	4	5

A la fin, `compt` vaut `5`, on a trouvé, dans l'ordre toutes les lettres du mot `satan`, le programme répond `True`.
Avec DÉMOCRATISATION ça finit moins bien.

lettre	d	e	m	o	c	r	a	t	i	s	a	t	i	o	n
Mot2[compt]	s	s	s	s	s	s	s	s	s	s	a	t	a	a	a
test	F	F	F	F	F	F	F	F	F	T	T	T	F	F	F
compt	0	0	0	0	0	0	0	0	0	1	2	3	3	3	3

Bilan : on regarde si on retrouve une à une les lettres de `Mot2` cachées dans `Mot1`, dans le bon ordre. C'est une sorte d'inclusion d'un mot dans l'autre.

On retrouve alors les prénoms cachés :

Prochain challenge : on donne les mots de la colonne de gauche à un programme et il cherche à retrouver le mot de la colonne de droite *(lettres communes à nos mots, mais en plus dans le même ordre, ça doit pas être mal à programmer...)*.

Le test
```
if compt==len(Mot2) :
....return(True)
```
est là pour quand, comme dans ELISA et RelUisaNT, on a égalité de la dernière lettre de `Mot2` (ici, le `a`) avec une lettre du milieu de `Mot2`. Si on ne sort pas à cet instant en rompant la boucle `compt` a atteint la valeur `len(Mot2)` (ici `5`) alors on fait le test `lettre==Mot2[5]` alors que `Mot2[5]` n'existe pas [11].

Le cadeau pour Solène : **LOUISE** et **EBLOUISSEMENT** , mais pour **ANAIS**, j'ai canabis, démangeaison, clef-anglaise, française, vandalisme, vantardisme, organisations, anachronisme, et même azerbaïdjanais si on veut un ï (ah non, pas au bon endroit).

Lycee Charlemagne I.T.C.

Le mot le plus long.

On va ouvrir le fichier en mode lecture simple et donner un nom au fichier ouvert. Ensuite, on le parcourera ligne à ligne.

```
import os
Dico = open('FRENCK.TXT','r') #simple mode lecture
for Mot in Dico : #comme ça on les aura tous
....oui tiens, on fait quoi ?
```

On va devoir tester si toutes les lettres d'un `mot` sont différentes.
Pour celà, on crée une procédure de test :

```
def PasDeLettreEnDouble(Mot) :
....for k in range(len(Mot)) : #on regarde les lettres une à une
........for i in range(k) : #on regarde les lettres qui précédent
............if Mot[k] == Mot[i] : #si il y a une égalité de lettres
................return(False) #on sort et on dit "pas bon"
....return(True) #il n'y a eu aucun doublon, c'est bonnard !
```

On crée ensuite une variable `Record`, qui va mémoriser la longueur du `mot` le plus long trouvé jusqu'à présent :

11. on rappelle que le dernier élément d'une liste de longueur n est L[n-1]

```
Record = 0
for Mot in Dico :
....if PasDeLettreEnDouble(mot) : #on ne regarde que les mots sans doublons
........if len(Mot) > Record : #chaque fois qu'on a trouvé mieux
............Record = len(Mot) #on mémorise la longueur
............MotRecord = Mot #on mémorise le mot
Dico.close() #par mesure de sécurité
```

On peut aussi se dire qu'on ne testera que les mots longs :

```
Record = 0
for Mot in Dico :
....if len(Mot) > Record : #déjà, il faut un nouveau record
........if PasDeLettreEnDouble(mot) : #chaque fois qu'on a trouvé mieux
............Record = len(Mot) #on mémorise la longueur
............MotRecord = Mot #on mémorise le mot
Dico.close() #par mesure de sécurité
```

Sinon, il faut aussi une procédure qui compte combien il y a de lettres différentes dans un mot.
On fait simple. Chaque fois qu'on nous donne un mot, on le recopie lettre à lettre ; mais quand une lettre a déjà été utilisée, on ne la prend pas :

```
def NbLettres(Mot) :
....NouveauMot = "
....for lettre in Mot :
........if not(lettre in NouveauMot) :
............NouveauMot += lettre
....return(len(NouveauMot))
```

Par exemple, avec `Mot` = 'PSYCHOLOGIQUEMENT', on arrive à `NouveauMot` = 'PSYCHOLOGIQUEMNT',
tandis qu'avec `Mot` = 'ANTICONSTITUTIONNELLEMENT', on a `NouveauMot` = 'ANTICOSUELM', ce qui est décevant.
Ensuite, on crée un parcours avec record qu'on pousse au fur et à mesure :

```
Dico = open('FRENCK.TXT','r')
record = 0
for Mot in Dico :
....if NbLettres(mot) > record :
........record = NbLettres(mot)
........MotRecord = Mot
print(MotRecord)
```

Lycee Charlemagne	I.T.C.
Crescendo decrescendo.	

On crée une procédure qui prend un mot et regarde si ses lettres sont classées par ordre croissant, décroissant ou change de sens de variation.

On mesure donc le signe de `Mot[k+1]-Mot[k]` et de `Mot[k+2]-Mot[k+1]`.
Si les deux sont de même signe, la suite est monotone.
Si les deux sont de signe opposé, la suite a un extremum.

	`Mot[k]<Mot[k+1]`	`Mot[k]>Mot[k+1]`
`Mot[k+1]<Mot[k+2]`	croissante	minimum local
`Mot[k+1]>Mot[k+2]`	maximum local	décroissante

Comme `Mot[k]<Mot[k+1]` et `Mot[k+1]<Mot[k+2]` sont des booléens, ils valent `0` ou `1`, le tableau se résume en

somme des booléens	True	False
True	2	1
False	1	0

et on constate que le changement de sens de variation se lit par `(Mot[k]<Mot[k+1])+(Mot[k+1]<Mot[k+2])==1`
(on pouvait aussi jouer avec un ou exclusif `(Mot[k]<Mot[k+1])^(Mot[k+1]<Mot[k+2])`).
On doit donc savoir si il y a trop de changements de sens de variation :

```
def MotCroissantDecroissant(Mot) :
....#if len(Mot)<2 : #à quoi bon tester les mots trop courts pour varier
....#....return(False)
....change=0 #on initialise
....for k in range(len(Mot)-2) : #on va aller regarder Mot(k+2) on s'arrête avant la fin
........if (Mot[k]<Mot[k+1])+(Mot[k+1]<Mot[k+2]) == 1 :
............change+=1 #on a détecté un extrémum local
....return(change<2) #il n'y a eu que 0 ou 1 changement de variation
```

Je vous connais, vous auriez conclus avec

```
....if change < 2 :
........return(True)
....else :
........return(False)
```

Mais quel gâchis ! `change<2` est déjà un booléen qui vaut `True` ou `False`. C'est comme vos test en if `(a==b) == True`. C'est mignon... et ridicule.

On n'a plus qu'à utiliser ce petit programme :

```
import os
Dico=open('French.txt','r')
L = [ ]
for Mot in Dico :
....if MotCroissantDecroissant(Mot) :
........L.append(Mot)
print(L)
Dico.close()
```

```
from os import *
Dico = open('FRENCH.TXT', 'r')
DicoMots = [ ] #pour les mots proprement écrits
DicoSorted = [ ] #pour les mots triés

for mot in Dico : #on lit le dictionnaire
....DicoMots.append(mot) #on remplit le dictionnaire des mots lisibles
....MotListe = sorted(list(Mot)) #on crée l'anagramme trié
....DicoSorted.append(MotListe) #on le mémorise
Dico.close() #on est propre, on ferme

for k in range(len(DicoSorted)) : #on va lire le dictionnaire trié
....Mot1 = DicoSorted[k] #on prend le mot pour ne pas le lire et le relire
....for i in range(k) : #on ne regarde que les mots qui précèdent
........Mot2 = DicoSorted[i] #on lit le mot
........if len(Mot1) == len(Mot2) : #on ne regarde que si ils ont la même longueur
............if Mot1 == Mot2 : #on compare les mots triés
................print(DicoMots[i], DicoMots[k]) #on affiche les mots sous forme lisible

print('Et voila, fini !') #on prévient qu'on a fait le tour
```

Lycee Charlemagne I.T.C.

Correcteur orthographique.

Transformation de FRENCH.TXT en Dico.

```
import os.
fichier = open('FRENCH.TXT', 'r')
Dico = [ ]
for mot in fichier :
....Dico.append(mot[ :-2])
fichier.close()
```

On ouvrira même avec **import os, time** car on aura besoin de **time** plus loin.
On n'oublie pas d'affecter l'ouverture à une variable.
De plus, dans l'ouverture, le nom de fichier doit être mis entre guillemets.
Il est d'autre part absurde de penser travailler sur des **FRENCH.TXT.readlines()** car **FRENCH.TXT** n'est pas un objet **Python**.
La variable **mot[:-2]** prend **mot** et ne le recopie que jusqu'à deux caractères de la fin, car dans le fichier ouvert, chaque mot contient deux caractères de "retour chariot" à la fin de chaque ligne.
On peut aussi lire le fichier entier **Dico = fichier.readlines()** et découper par la méthode **D.split()**, le caratère de découpage étant justement le **'/n'** de retour chariot.

Distance (et étapes) de SOLENE à NICOLAS.

On se contente de reprendre en sens inverse le chemin de NICOLAS à SOLENE.
On disposait de

nicolas	$R_{0,0}$	sicolas	E_1	scolas	E_1	solas	$R_{3,3}$	soles	$R_{4,4}$	solen	$I_{5,5}$	solene

On peut créer

solene	E_5	solen	$R_{4,6}$	soles	$R_{3,5}$	solas	$I_{1,2}$	scolas	$I_{1,1}$	sicolas	$R_{0,0}$	nicolas

Il faudrait quand même prouver qu'il n'y a pas de chemin plus court. Il faudrait argumenter sur le nombre de lettres différentes déjà.
Il n'y a pas unicité du chemin de longueur 6.

SOLENE	R	NOLENE	E	NOLEE	R	NOLAE	R	NOLAS	I	NIOLAS	I	NICOLAS
SOLENE	R	SOLENS	R	SOLEAS	E	SOLAS	I	SIOLAS	I	SICOLAS	R	NICOLAS

Ensuite, je peux me demander si des E_i commutent avec des $R_{j,k}$.

Distance (et étapes) de SUCRI à CHRIST.

On peut faire ça

`SUCRI`	E_0	`UCRI`	E_0	`CRI`	$I_{1,1}$	`CHRI`	$I_{4,4}$	`CHRIS`	$I_{5,5}$	`CHRIST`

Et j'avoue bien humblement ne pas pouvoir faire plus court.
Il faut au moins effacer ou remplacer `S` et `U`. Et ajouter `H`, `S` et `T`.
On me dira qu'il est idiot d'effacer un `S` pour en remettre un. Mais il n'est pas au bon endroit. De plus, on ne peut pas gagner du temps en remplaçant le U par un `T` par exemple ; les emplacements ne sont pas les bons. Ce qui est utile, c'est un $I_{k,k}$.

Distance (et étapes) de CHIEN à NICHE.

Là aussi, les solutions sont multiples :

`CHIEN`	E_2	`CHEN`	E_3	`CHE`	$I_{0,1}$	`ICHE`	$I_{0,0}$	`NICHE`
`CHIEN`	I	`NCHIEN`	I	`NICHIEN`	E	`NICHEN`	E	`NICHE`

On ne pourra pas faire plus court, car si on utilise un E, il faudra utiliser une I, et on ne peut pas n'utiliser que des R (sauf à en utiliser 5 car aucune lettre n'est au bon endroit attendu).
Je sais, ce n'était pas évident, il fallait trouver une séquence ordonnée présente dans les deux : le CHiEn et niCHE.

Compactage du bloc en une ligne.

La variable crée est `cout`, elle vaut `0` ou `1`, suivant que `MotA[i-1]` est égal à `MotB[j-1]` ou pas.
On peut profiter du fait qu'un booléen vaut `0` pour `False` et `1` pour `True`.
On proposera `cout = (MotA[i-1] != MotB[j-1])`.
Ou `cout = 1-(MotA[i-1] == MotB[j-1])` *(ici, le fait de faire un calcul transforme vraiment le booléen en entier)*.

Exécution de Leven('SUCRI', 'CHRIST').

Le bloc
```
print('Etape ', i)
for k in range(LA+1) :
....print(D[k])
```
affiche le texte Etape et l'indice i (qui varie de 1 à LA (inclus) puis affiche ligne par ligne la matrice D

La matrice D a été initialisée par

`D = [[0 for j in range(LB+1)] for i in range(LA+1)]`

elle a `LA+1` lignes, formées de `LB+1` termes.

`LA` est la longueur du premier mot passé *(ici 'Sucri' LA=5)*, et `LB` est la longueur du second mot passé *(ici 'Christ' LB=6)*

Les deux `+1` ne sont pas des erreurs, ni des compensations du principe informatique des `range` ; la matrice est bien plus grande que les mots traités.

Ici, `D` a `6` lignes de longueur `7`.

La matrice initiale est truffée de `0`.

$$\begin{pmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{pmatrix}$$

Mais ensuite, on modifie quelques termes

```
....for i in range(LA+1) :
........D[i][0]=i
```

les termes d'indices `0` de chaque ligne valent `i` *(indice de ligne)*

```
....for j in range(LB+1) :
........D[0][j]=j
```

les termes de la ligne d'indice 0 valent j *(indice de colonne)*.

$$\begin{pmatrix} 0&1&2&3&4&5&6\\ 1&0&0&0&0&0&0\\ 2&0&0&0&0&0&0\\ 3&0&0&0&0&0&0\\ 4&0&0&0&0&0&0\\ 5&0&0&0&0&0&0 \end{pmatrix}$$

La première matrice est

Ensuite, on modifie ligne par ligne la matrice, en la parcourant, avec des tests plus ou moins clairs. J'applique :

0	1	2	3	4	5	6
1	1	2	3	4	4	5
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

0	1	2	3	4	5	6
1	1	2	3	4	4	5
2	2	2	3	4	5	5
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

0	1	2	3	4	5	6
1	1	2	3	4	4	5
2	2	2	3	4	5	5
3	2	3	3	4	5	6
4	0	0	0	0	0	0
5	0	0	0	0	0	0

0	1	2	3	4	5	6
1	1	2	3	4	4	5
2	2	2	3	4	5	5
3	2	3	3	4	5	6
4	3	3	3	4	5	6
5	0	0	0	0	0	0

0	1	2	3	4	5	6
1	1	2	3	4	4	5
2	2	2	3	4	5	5
3	2	3	3	4	5	6
4	3	3	3	4	5	6
5	4	4	4	3	4	5

Je vous laisse comprendre sur cet exemple quel mot est formé peu à peu pour passer de `sucri` à `christ`.

Dans l'instruction `D[i][j]=min(D[i-1][j]+1, D[i][j-1]+1, D[i-1][j-1]+cout)`

on reconnaît peut être les trois types d'étapes de la méthode de Levenshtein :

effacement, insertion, remplacement

Mais il faut vraiment se prendre la tête.

Recherche des mots proches.

```
def MostProches(Mot) :
....Tdebut = time.clock()
....if Mot in Dico :
........return(True, 0, time.clock()-Tdebut)
....Mini = Leven(Mot, Dico[0])
....MotsProches = [Dico[0]]
....for BonMot in Dico :
........Dist = Leven(Mot, BonMot)
........if Dist < Mini :
............Mini = Dist
............MotsProches = []
........id Dist == Mini :
............MotsProches.append(BonMot)
....return(False, Mini, MotsProches, time.clock()-Tdebut)
```

Avec `Tdebut`, on mémorise l'heure de début, et avec `time.clock()-Tdebut`, on mesure donc le temps écoulé entre l'entrée dans la procédure, et la sortie.
Les comparaisons etre `Mini` et `Dist` sont la classique recherche d'un *minimum (on a initialisé avec le premier mot du dictionnaire)*.
La liste `MotsProches` est un tampon que l'on remet à zéro quand la distance diminue *(`BonMot` est un record)*, et qu'on agrandit quand la distance stagne *(`BonMot` fait aussi bien que le record précédent)*, et à laquelle on ne fait rien quand `BonMot` n'est vraiment pas mieux que le record.

Procédure Trigrammes.

Un mot de n lettres a n trigrammes.
En effet, pour chaque lettre du mot jusqu'à l'avant dernière, il y a un trigramme (la lettre et les deux lettres qui suivent, et aussi pour l'avant dernière lettre, il y a deux lettres suivies du symbole `$`). Et il y a aussi un trigramme initial formé de `$` suivi des deux premières lettres du mot.
Pour n lettres, il y a `n-1` trigrammes purs en `def`, un trigamme d'ouverture `$ab` et un trigramme de cloture `yz$`.

```
def Trigrammes(Mot) :
....mot = '$'+Mot+'$'
....Trig = []
....for k in range(len(mot)-2) :
........Trig.append(mot[k :k+3])
....return(Trig)
```

On travaille sur `mot` (sans majuscule) pour ne pas abimer `Mot` (avec majuscule), et en en profite pour coller les deux $ typographiques.
La liste des trigrammes est initialisée à une liste vide qu'on agrandira.
Le range va de `0` à `len(mot)-2`, pour que l'extraction `mot[k :k+3]` s'arrête à la fin de `mot`.
On peut construire la boucle implicitement :

```
....Trig = [mot[k :k+3] for k in range(len(mot)-2)]
```

On peut aussi lire les lettres une à une :

```
....mot = '$'+Mot+'$'
....L1, L2 = Mot[0], Mot[1]
....Trig = []
....for k in range(2 : len(mot)) :
........L1, L2, L3 = L2, L3, mot[k]
........Trig.append(L1+L2+L3)
```

Dictionnaire MPSI2.

1- Que donne `MPSI['Maths'].remove('Choquet')` ?
On va chercher le liste MPSI['Maths'], qui existe, et de cette liste, on efface 'Choquet'. Dans cette instruction, on n'appelle pas d'élément qui n'existe pas.
Attention, si on l'exécute deux fois, on a un message d'erreur, puisque la liste MPSI['Maths'] ne contient plus 'Choquet'. (déjà parti à la retraite ?)

2- Quelle instruction devez vous taper pour ajouter l'informatique et l'I.P.T.
```
MPSI['Informatique']=['Brunet']
MPSI['ITC']=['Brunet', 'Choquet', 'Chabane']
```
Il faut deux instruction.

3- Quelle instruction devez vous taper pour que madame CHEVALIER s'appelle CHEVALIER-THERY ?
On peut certes remplacer la liste :
```
MPSI['Physique'] = ['Chevalier-Thery']
```
Mais comme on peut la trouver sur d'autres matières :
```
for matiere in MPSI.keys() :
....Profs = MPSI2[matiere] :
....for k in range(len(Profs)) :
........if Profs[k] == 'Chevalier' :
............Profs[k] = 'Chevelier-Thery'
```

4- Quel est l'effet de **for A in MPSI : MPSI[A].append('Sucri')** ?
A chaque liste de MPSI, on ajoutre Sucri. Il devient prof omnipotent...

5- La `MPSIn` a aussi un dictionnaire du même type. Tapez une instruction qui affecte l'I.P.T. à la réunion des professeurs de maths et de physique, si toutefois il n'y a pas déjà un prof d'I.P.T. dans le dictionnaire.
```
if not('ITC' in PCSI.keys()) :
....PCSI['ITC']=['Brunet']+PCSI['Maths']+PCSI['Physique']+PCSI['Chimie']
```
Et peut être même :
```
Matieres = PCSI.keys()
if not('IPT' in Matieres) and 'Maths' in Matieres and 'Physique' in
Matieres and 'Chimie' in Matieres :
....PCSI['IPT']=[Brunet]+PCSI['Maths']+PCSI['Physique']+PCSI['Chimie']
```
afin de s'assurer qu'il y a bien des `Maths` et de la `Physique` dans cette PCSI...
Tiens, mais si c'est le même prof en maths, en physique et en chimie ? Non, ne parlez pas d'horreur.

Citation de Proust.

La variable `texte` contient une chaine de caractères qu'on va explorer lettre par lettre *(y compris les espaces et la ponctuation)*.
On initialise un dictionnaire au sens pythonien du terme, qui s'appelle `table` *(on reconnaît au fait qu'on initialise avec des accolades)*.
Pour chaque lettre, on regarde si elle est déjà dans les clefs de la `table`.
Si ce n'est pas le cas, on la crée comme clef, avec comme élément associé l'entier 1.
Peu à peu, `table` contient des couples (`lettre, 1`).
On a donc `{'l' : 1}`, puis `{'l' : 1, 'o' : 1}`, puis `{'l' : 1, 'o' : 1, 'n' : 1}` et ainsi de suite, jusqu'à `{'l' : 1, 'o' : 1, 'n' : 1, 'g' : 1, 't' : 1, 'e' : 1, 'm' : 1, 'p' : 1, 's' : 1, ' ' : 1, 'j' : 1}`
La lettre suivante est un `'e'` encore. Il est déjà dans `table.keys()`. On sollicite alors `table['e']` et on l'augmente de `1` :
`{'l' : 1, 'o' : 1, 'n' : 1, 'g' : 1, 't' : 1, 'm' : 1, 'p' : 1, 's' : 1, ' ' : 1, 'j' : 1, 'e' : 2}`
Pour chaque symbole nouveau, on met un compteur à `1`.
Pour chaque symbole déjà croisé, on augmete son compteur de une unité.
On crée donc un histogramme des lettres de la phrase.
`{' ' : 7, 'c' : 2, 'b' : 1, 'e' : 7, 'é' : 1, 'd' : 1, 'g' : 1, 'i' : 1, 'h' : 2, 'j' : 1, 'm' : 2, 'l' : 1, 'o' : 3, 'n' : 3, 'p' : 1, 's' : 3, 'r' : 1, 'u' : 3, 't' : 1}`
Il ne faut pas avoir oublié `' ' : 7`.

Création du dictionnaire BONSTRIGRAMMES.

```
BonsTrigrammes = { }
for Mot in Dico :
....Trigs = Trigrammes(Mot)
....for trig in Trigs :
........if trig in BonsTrigrammes.keys() :
............BonsTrigrammes[trig].append(mot)
........else :
............BonsTrigrammes[trig] = [mot]
```

Les programmeurs pourront même écrire

```
for mot in Dico :
....for trig in Trigrammes(mot) :
........BonsTrigrammes[trig]=BonsTrigrammes.get(trig, [])+[mot]
```

En effet, la méthode `get` appliquée aux dictionnaires est quelquechose de bien pratique :
`dict.get(a, sinon)` cherche si il y a dans `dict` une clef appelée `a`
• si il y en a une, on prend `dict[a]`
• si il n'y en a pas, on crée la valeur `sinon`.
Ici, donc, avec `BonsTrigrammes[trig]=BonsTrigrammes.get(trig, [])+[mot]`
• si `trig` est une clef de `BonsTrigrammes`, alors `BonsTrigrammes.get(trig, [])` est `BonsTrigrammes[trig]`, et on ajoute `[mot]` *(par addition, la liste s'agrandit)*
• si `trig` n'est pas une clef de `BonsTrigrammes`, alors `BonsTrigrammes.get(trig, [])` est égal à `[ ]` *(liste vide)*. On lui ajoute `mot`, on obtient `[mot]` et on l'affecte à `BonsTrigrammes[trig]`. qui n'existait pas encore.
Génial, n'est il pas ?

Indice de Jacquard.

Que signifie $Jac(A,B) = 0$? A et B ont une intersection vide. Ils sont disjoints.
Que signifie $Jac(A,B) = 1$? Le cardinal de $A \cap B$ est égal à celui de $A \cup B$. Comme l'un est inclus dans l'autre, c'est qu'ils sont égaux. Comme on a $A \cap B \subset A \subset A \cup B$, on arrive à $A = A \cup B$, puis $B = A \cup B$. Comme on s'en doutait : A est égal à B.
Que signifie $Jac(A,B) = 2$? Que le cardinal de $A \cap B$ est plus grand que celui de $A \cup B$. Ce n'est pas possible.
A moins que l'on ne prenne l'ensemble vide, mais $\frac{Card(\emptyset \cap \emptyset)}{Card(\emptyset \cup \emptyset)}$ n'a pas de sens...

Mots d'indice plus grand que 0.2.

```
#liste des trigrammes de Mot
Trig=Trigrammes(Mot)
#creation d'un dictionnaire des mots ayant au moins un trigramme commun avec Mot
#on prend les trigrammes de Mot, et on regarde à quels bons mots du dictionnaire ils renvoient
Commun={ }
for trig in Trig :
....for MotCorrect in BonsTrigrammes[trig] :
........if MotCorrect in Commun.keys() :
............Commun[MotCorrect] += 1
........else :
............Commun[MotCorrect] = 1
#maintenant, on utilise Commun, et on regarde quels mots ont un bon taux
MotsProches = [ ]
for MotCorrect in Commun :
....NbTrig = float(Commun[MotCorrect])
....taux = NbTrig/(len(MotCorrect)+len(MotA)-NbTrig)
....if taux > 0.2 :
........MotsProches.append(MotCorrect)
```