Cet article est rédigé par des élèves. Il peut comporter des oublis et imperfections,
autant que possible signalés par nos relecteurs dans les notes d'édition.

Le correcteur d'orthographe

Année 2022 - 2023

Dorian Adam, Valentin Amorim, Maxime Scotto di Rinaldi, Virgile Gaillard, Youssef Barrek,
élèves de première

Établissement : Lycée Français Vincent van Gogh (La Haye)
Professeurs : Stéphane Beringue, Line Boissonnet, Mathieu Buchwald, Florence Decool
Chercheuse : Marie Anastacio, Rheinisch-Westfälische Technische Hochschule Aachen

Sommaire

1. Problématique

Sujet

Lorsqu'un mot est incorrectement écrit, comment décider par quel mot le remplacer ?

2. Introduction

De nos jours, à cause des téléphones portables, les personnes tapent de plus en plus vite et font donc de plus en plus de fautes d'orthographe.

C'est pour cela que nous avons opté pour le sujet qui nous a été proposé : celui du correcteur d'orthographe.

3. Premières heures

Brainstorming

Durant nos premières heures de travail, nous avons réfléchi tout d'abord à différentes questions. Dans un premier temps nous nous sommes demandé s'il fallait plutôt réaliser un correcteur de grammaire ou d'orthographe et lequel des deux serait le plus adapté au codage.

Par la suite nous nous sommes demandé sur quels paramètres travailler, comme le nombre de lettres dans le mot ou ce que nous appellerons par la suite le "score" d'un mot.

Enfin, nous nous sommes demandé quel langage de programmation {Python, Java, C++...) serait le plus approprié et lequel aurait la meilleure efficacité pour notre projet.

Complexités du sujet

Par la suite, nous avons trouvé plusieurs difficultés pour notre sujet :

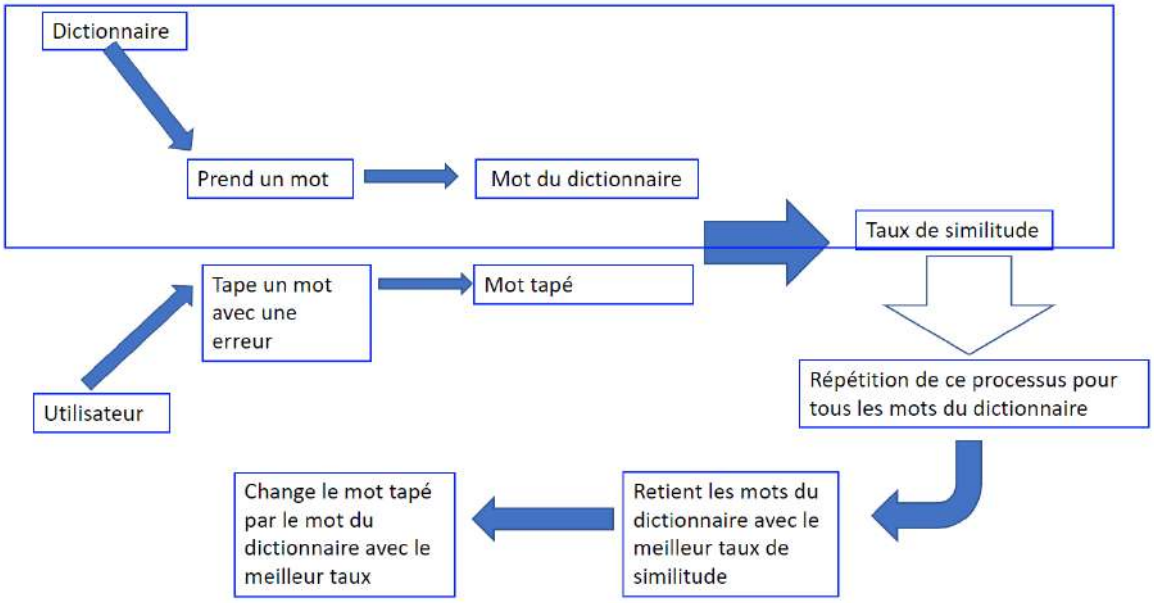
- Dans un premier temps, le fait qu'il y ait près de 32 000 mots dans le français courant.
- Puis, le fait qu'il y ait 26 lettres différentes dans l'alphabet (sans compter les accents, les apostrophes, la ponctuation etc.)
- Enfin, nous avons dû apprendre à coder en Python pour pouvoir réaliser des prototypes de nos idées.

Idée retenue

Nous avons fixé nos idées et nous nous sommes alors mis en tête que nous devions faire un correcteur d'orthographe et non de grammaire, car ce dernier serait trop compliqué à réaliser avec le temps mis à notre disposition si nous comptons toutes les exceptions, les conjugaisons *etc.*

Un correcteur d'orthographe sera donc plus facile à mettre en œuvre.

Le fonctionnement d'un correcteur d'orthographe selon nous



Ceci est un schéma du fonctionnement d'un correcteur d'orthographe. Selon nous un correcteur d'orthographe part de deux points :

- un dictionnaire (qui est une liste de tous les mots de la langue française sous toutes les formes)
- un utilisateur (qui va entrer un mot avec une/des erreur(s) potentielle(s).

- Le **dictionnaire** désigne une base de données qui va permettre de proposer un mot juste.
- Notre correcteur d'orthographe va d'abord choisir un par un les mots de ce dictionnaire. Pour chacun des mots du dictionnaire, le correcteur va, à l'aide d'une méthode de comparaison (déterminée par la suite), établir un taux de similitude ou une distance entre le mot du dictionnaire et le mot tapé par l'utilisateur.
- Il répète ce processus jusqu'à ce que chacun des mots du dictionnaire soient analysés.
- Une fois tous les taux de similitude déterminés, le correcteur va choisir les mots du dictionnaire qui ont obtenu les plus hauts taux de similitude (ou la plus faible distance) avec le mot tapé. Enfin, le correcteur propose à l'utilisateur ces mots.

La question maintenant est de trouver comment obtenir une formule pour déterminer un taux de similitude ou une distance entre deux mots.

Problèmes trouvés après brainstorming

Nous avons trouvé plusieurs problèmes suite à notre brainstorming, qui nous permettraient d'approfondir nos recherches :

- Comment calculer la distance entre 2 mots ?
- Comment devrions-nous procéder afin de corriger un mot mal tapé sur le clavier ?
- Quels sont les mots que nous devrions suggérer à l'utilisateur ?

4. Essais de code

4.1 Choix du langage de programmation

Nous avons opté pour le langage de programmation Python. En effet, il est adapté à notre niveau car nous avons déjà quelques bases.

Buts du programme

```
list = ['eau', 'vie', 'tel', 'rue', 'écu', 'élu', 'les', 'des', 'ver', 'met
moa = "eau"
mob = "vie"
moc = "tel"
mod = "rue"
moe = "écu"
mof = "élu"
mog = "les"
moh = "des"
moi = "ver"
moj = "met"
word=input("Taper le mot")
char1 = word[0]
char2 = word[1]
char3 = word[2]
mot = char1+char2+char3
if mot in list:
    print("Le mot est juste, pas de correction nécessaire!")
if char1 + char2 + char3 not in list:
    print("Le mot n'existe pas, voici les suggestions")
if char1 in moa:
    print(moa)
if char1 + char2 + char3 not in [moa,mob,moc,mod,moe,mof,mog,moh,moi,moj]:
    if char1 in [moa,mob,moc,mod,moe,mof,mog,moh,moi,moj]:
      sorted(moa,mob,moc,mod,moe,mof,mog,moh,moi,moj)
      print(sorted(moa,mob,moc,mod,moe,mof,mog,moh,moi,moj))
    elif char2 in [moa,mob,moc,mod,moe,mof,mog,moh,moi,moj]:
      sorted(moa,mob,moc,mod,moe,mof,mog,moh,moi,moj)
      print(sorted(moa,mob,moc,mod,moe,mof,mog,moh,moi,moj))
```

Nous avons écrit un programme se basant sur une liste d'une dizaine de mots.

Le but du programme est de déterminer des mots similaires et de suggérer les mots les plus cohérents à l'utilisateur dans une liste ordonnée. (1)

Lien du programme : https://replit.com/@MaximeSdR/MEJ3

Défauts du programme

Le programme donne une liste de mots, mais celle-ci ne permet pas d'obtenir les mots les plus proches du mot tapé.

La langue française est composée de plus de 32 000 mots, il est impossible de définir chacun des mots individuellement à grande échelle.

4.2 D'autres pistes de recherche

La première formule de taux de similitude que nous avons réalisé est la suivante :

$$\boldsymbol{Taux\ de\ similitude} = \frac{\boldsymbol{Nombre\ de\ lettres\ similaires}}{\boldsymbol{Nombre\ de\ lettres\ total}}$$

Alors, pourquoi cette formule n'a-t-elle pas marché ? (2)

Si on prend par exemple le mot « pomme » du dictionnaire, on obtiendrait alors ceci :

p|o|m|m|e
1|2|3|4|5|6
=1/6

mot tapé: p|p|o|m|m|e
1|2|3|4|5|6|7|

Le mot tapé par l'utilisateur étant « ppomme », la formule n'obtiendrait alors qu'un taux de similitude de 1/6 pour le mot pomme. Il ne proposerait alors pas ce mot qui est pourtant le plus évident pour nous.

5. Méthode des combinaisons

Dans les grandes lignes

La "méthode des combinaisons" est une manière de calculer un taux de similitude entre deux mots. Nous avons représenté cette méthode sous forme d'un organigramme qui proposera à la fin les mots au taux le plus élevé.

Il reprend notamment les principes du tableau dans la partie **3**. (Qu'il est conseillé de revoir avant de lire la suite)

Cette méthode est séparée en 2 façons de calculer ce taux, la première façon fonctionne à partir des lettres similaires et la deuxième façon fonctionne à partir des "combinaisons de lettres". Nous préciserons le fonctionnement en détails plus tard.

La méthode et définitions

Cette formule est inspirée de la première formule qu'on avait trouvée (celle qui comparait les positions des lettres similaires du mot tapé et du mot du dictionnaire et divisait le tout par le nombre de lettres dans le mot du dictionnaire). Mais, pour cette formule nous ne prenons plus en compte les positions des lettres (ce qui nous débarrasse du défaut de la première formule).

Il faut maintenant définir certains termes utilisés dans notre explication :

découpage : étape de l'organigramme où un mot séparé en plusieurs combinaisons

combinaison : un groupe de deux lettres - un singleton : lorsqu' une lettre se retrouve dans une combinaison sans autre lettre

Un singleton : lorsqu'une lettre va se retrouver sans binôme

Comment obtenir ces fameuses combinaisons ?

Le découpage du mot tapé et du mot du dictionnaire doit être très rigoureux et doit d'abord respecter certaines règles car sinon la méthode ne marcherait pas : les combinaisons ne peuvent qu'être composées de lettres côte à côte.

Il faut aussi prendre en compte la longueur et si elle est impaire ou paire :

- Si le mot est pair, le découpage sera simple à faire, les lettres seront groupées 2 par 2 (ex : mathématique -> ma+th+ém+at+iq+ue).
- Si le mot est impair, on devine qu'une lettre se retrouvera forcément seule, on l'appelle singleton. Il faut que le singleton ait uniquement une position impaire car sinon le découpage crée trop de combinaisons susceptibles de modifier gravement le résultat final.

Au final, le découpage d'un mot impair nous donne plusieurs combinaisons (**p**+en+sa+it pe+**n**+sa+it pe+ns+**a**+it pe+ns+ai+**t**) où l'on retrouve des singletons comme p, n, a, t et l'on remarque que tous ont une position impaire (1, 3, 5, 7). (3)

Si le singleton a une position paire cela va engendrer la création d'autres singletons et donc de combinaisons comme dans cet exemple où "e" se retrouve sans binôme : **p**+e+ns+ai+**t.** P et T sont des singletons en excédent et engendrent alors un taux qui a pour dénominateur x/5 (car on compte 5 combinaisons) au lieu de x/4.

Dernière précision, les mots du dictionnaire à analyser seront ceux qui ont 0, 1 ou 2 lettres de plus ou de moins que le mot tapé. Cette étape a pour unique but d'optimiser l'algorithme mais peut être perçue comme facultative.

Calculs

La méthode est fondée sur deux formules assez simples. Ces formules vont déterminer un taux de ressemblance entre deux mots :

- La première fonctionne en divisant les combinaisons de lettres qu'ont en commun les deux mots par le nombre de combinaisons qu'a le mot proposé par le dictionnaire.
- La deuxième est assez similaire, on divise le nombre de lettres qu'ont en commun les deux mots par le nombre total de lettres dans le mot du dictionnaire.

Les deux formules s'appliqueront en fonction du nombre de caractères dans le mot tapé : la première s'applique lorsque le mot tapé a plus de 4 caractères, et la deuxième s'applique si le mot a moins de 4 caractères.

Explications des choix :

Nous avons décidé de séparer les mots en deux catégories pour plusieurs raisons :

- La première raison est que cela permet déjà à l'algorithme d'analyser moins de mots, car l'algorithme ne va pas analyser les mots de 13 lettres qui n'ont rien à voir avec le mot tapé.
- La deuxième raison est que chaque mot soit analysé avec la formule adaptée. En effet, si un mot tapé (≤4) est découpé en combinaisons, cela donnera trop peu de combinaisons potentiellement similaires et donc trop peu de résultats différents.
- Exemple : case-> ca+se --> on a que 2 combinaisons, peu de résultats possibles : 0%, 50%, ou 100%
- Alors que : c+a+s+e --> on a 4 combinaisons, plus de résultats possibles : 0%, 25%, 50%, 75%, 100%

Pourquoi avoir choisi de découper les mots en combinaisons ?

- Un mot tapé trop long découpé ainsi donnerait trop de combinaisons, et donc l'algorithme sera susceptible de proposer des anagrammes du mot tapé (Ex d'anagramme : pensait -->p+e+n+s+a+i+t --> inaptes). Or on ne veut pas que l'algorithme propose des mots trop différents du mot tapé. Pour y remédier nous utilisons des combinaisons de lettres. Nous allons nous demander comment obtenir ces combinaisons.

Exemple

Nous allons suivre les étapes de cet organigramme : (4)

https://lucid.app/lucidspark/1060e8dc-d3df-470e-8819-52ee72afad4d/edit?viewport_loc=-4980%2C6957%2C19192%2C9266%2C0_0&invitationId=inv_4507a66a-51cb-4c67-b41c-2b699878cf85

Si nous prenons comme exemple de mot tapé : “penssait” et que nous prenons comme exemple de mot du dictionnaire “pensait” et “passait” qui sont deux mots assez similaires. Après avoir suivi toutes les étapes de l'organigramme, il nous proposera ces résultats

(Consultez ce Powerpoint pour voir les étapes détaillées qui mènent jusqu'à ces résultats : https://docs.google.com/presentation/d/1zD0fIN_4_0UE4w1iTLLjfw-wkTkFmSm4HeWVHGz2I-s/edit?usp=sharing) (5)

Mot tapé : pe+ns+sa+it et Mot du dictionnaire : pa+ss+ai pas de combinaisons identiques et 3 combinaisons du mot dictionnaire au total donc le résultat est de 0/3 soit 0% de ressemblance.

Mot tapé : pe+ns+sa+it et Mot dictionnaire : pe+ns+a+it 3 combinaisons identiques et 4 combinaisons du mot dictionnaire au total donc le résultat est de 3/4 soit 75 %.

En conclusion, "pensait" est plus approprié pour corriger "penssait" car ce dernier a un taux de similitude plus élevé que "passai" : 75% est supérieur à 0%.

On peut maintenant éclairer les limites et les points forts de l'algorithme comme le fait que si l'utilisateur fait trop de fautes dans son mot et donc l'algorithme ne sera potentiellement pas capable de trouver le mot juste.

Cependant, l'algorithme est très efficace pour corriger les fautes où la lettre est remplacée, retirée ou ajoutée et l'on peut même préciser où est l'erreur dans le mot grâce au singleton puisqu'il est souvent à proximité de l'erreur commise. On peut aussi imaginer une formule hybride qui combinerait les deux méthodes de calculs.

6. Distance de Levenshtein

Définition de la distance de Levenshtein

On définit la distance de Levenshtein comme le nombre minimum d'opérations à effectuer pour passer du mot tapé à un mot du dictionnaire dans notre cas.

On distingue 3 opérations différentes, toutes ayant la même influence sur la distance :

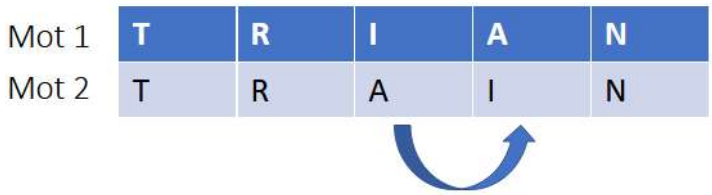
- L'ajout d'une lettre **(insertion)**
- La suppression d'une lettre **(suppression)**
- La substitution d'une lettre par une autre **(substitution)**

Exemple simple et définition de notation :

Prenons par exemple le mot "TRIAN" comme étant le mot tapé par l'utilisateur. Comparons-le au mot "TRAIN", qui semble être un mot proche.

Pour passer de "TRIAN" à "TRAIN", il faut effectuer 2 opérations :

- Une première **substitution** (on remplace le I par un A)
- Une seconde **substitution** (on remplace le A par un I).



On dit alors que la **distance de Levenshtein** entre les mots "TRIAN" et "TRAIN" vaut 2, car on effectue **au minimum 2 opérations** pour passer d'un mot à l'autre. On note :
lev(TRIAN,TRAIN) = 2

Exemple plus en profondeur

Etudions un autre cas, cette fois-ci de façon plus concrète.

On considère le mot tapé comme étant "PMOME". Nous pouvons alors relever quelques caractéristiques de ce mot :

- Il est composé de 5 caractères.
- Il commence par la lettre P.
- Ce n'est pas un mot du dictionnaire.

Nous pouvons alors procéder à la recherche de mots proches auxquels comparer ce mot.

Mots proches

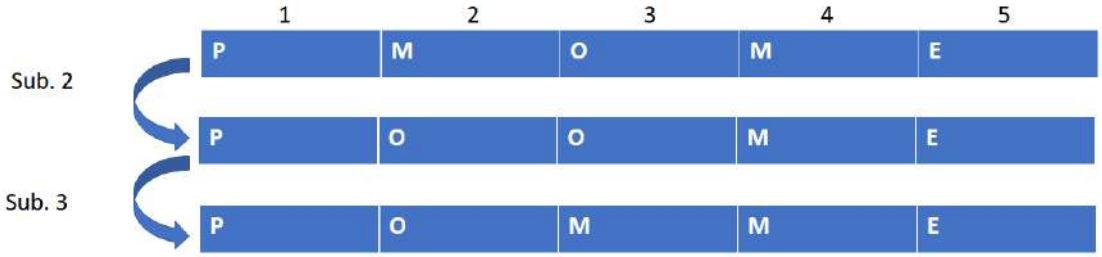
En cherchant dans le dictionnaire, on trouve facilement plusieurs mots proches, notamment :

- POMME
- POMMIER
- PAUME

Ces mots semblent quelque peu tous avoir une orthographe similaire, et semblent tous être de la même famille.

Calculons alors la distance de Levenshtein entre ces 3 mots et "PMOME", et déterminons quel mot suggérer à l'utilisateur.

Exemple de POMME



Prenons le mot « POMME » comme exemple.

Pour calculer la distance de Levenshtein entre ce mot et PMOME, il faut vérifier et déterminer le nombre d'opérations **le plus faible** pour passer d'un mot à l'autre.

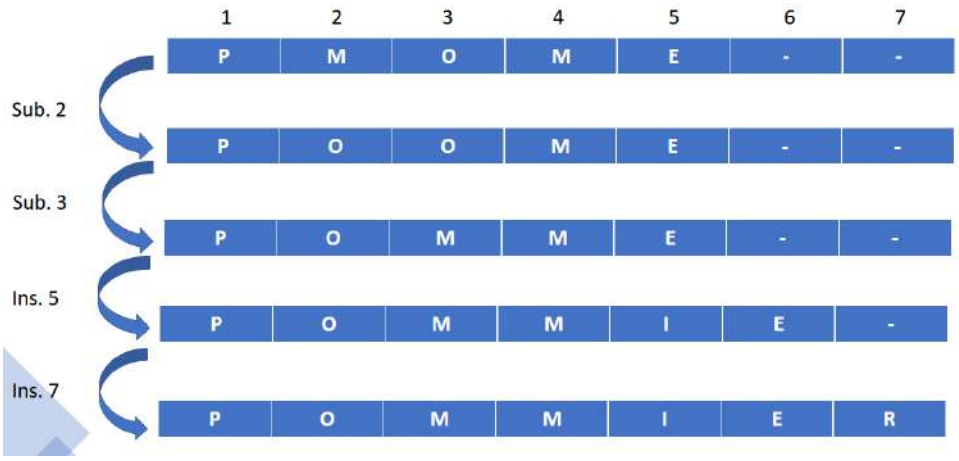
En l'occurrence, dans ce cas-ci, il faut seulement effectuer deux opérations pour passer de PMOME à POMME :

- Une première substitution de la lettre M pour un 0
- Une seconde substitution de la lettre M pour un 0

Deux étapes ont été effectuées, on dit alors que la distance de Levenshtein entre PMOME et POMME vaut 2.

On note alors: lev(PMOME,POMME) = 2 (2 substitutions)

Exemple de POMMIER



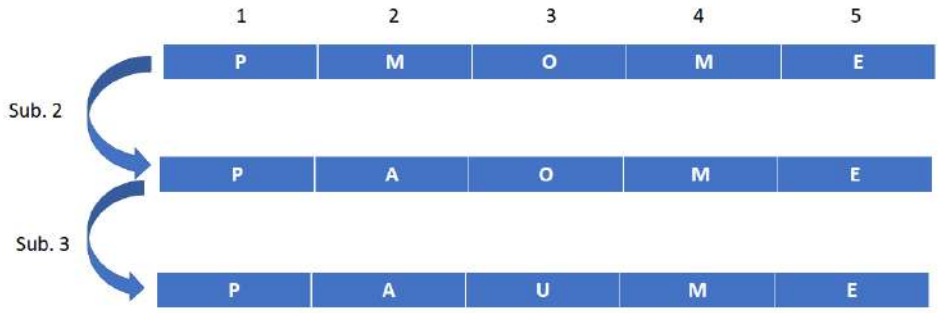
Pour l'exemple de POMMIER, on procède de la même manière.
On peut passer par POMME pour obtenir POMMIER : on obtient alors 2 opérations déjà effectuées.

Puis, pour passer de POMME à POMMIER, il suffit d'effectuer deux opérations supplémentaires :

- Une insertion d'un I à la position 5
- Une insertion d'un R à la position 7.

On a effectué 4 étapes, on dit alors que la distance de Levenshtein entre PMOME et POMMIER vaut 4.
On note alors : lev(PMOME,POMMIER) = 4 (2 substitutions+ 2 insertions) (6)

Exemple de PAUME



Comme nous avons effectué 2 étapes, nous pouvons en déduire que :

lev(PMOME,PAUME) = 2 (2 substitutions)

Introduction à la méthode matricielle

Pour calculer la distance de Levenshtein entre deux mots, nous pouvons aussi utiliser des matrices et des chaines de caractères, ainsi que leurs caractéristiques.

Pour employer cette méthode correctement, il faut tout d'abord définir les deux chaines de caractères principales par leur longueur totale (aussi appelée cardinal) :

$|a|$: Longueur totale de la chaine a

$|b|$: Longueur totale de la chaine b

Pour calculer la taille de la matrice, on considère qu'elle est de taille |a| + 1 x |b| + 1.

On définit une matrice comme un tableau de nombres réels.

Visualisation de la matrice

Prenons un exemple, avec la chaine *a* "POMME" et la chaine *b* "PAUME".

Les chaines PMOME et PAUME ont toutes les deux 5 caractères.
Or, il faut une matrice de taille m + 1 x *n* + 1, soit 5+1 par 5+1. On obtient donc forcément une matrice de taille 6 x 6.

----------	P	A	U	M	E
P					
O					
M					
M					
E					

À quoi sert exactement cette matrice ?

Tout d'abord, la matrice sert à calculer la distance de Levenshtein entre deux chaines de caractères, c'est **son but principal.**

Pour arriver à la distance de Levenshtein finale, la matrice nous permet de calculer des **"sous--distances",** entre des chaines de caractères plus courtes (marquées par toutes les cases vides, sauf la dernière).

La distance de Levenshtein finale entre les deux chaines de caractères principales est obtenue dans **la case en bas à droite,** encadrée en rouge.

----------	P	A	U	M	E
P					
O					
M					
M					
E					

Définition de *i* et *j*

Définissons maintenant deux variables *i* et *j* qui concernent la longueur des chaines *a* et *b* respectivement :

i : Variable de la longueur de la chaine a dans la matrice telle que $0 \leq i \leq |a|$
(i premiers caractères de la chaine)

j : Variable de la longueur de la chaine b dans la matrice telle que $0 \leq j \leq |b|$
(j premiers caractères de la chaine)

Pour illustrer ceci, on considère une chaine de caractère *a* "POMME", et une chaine de caractère *b* « PAUME» :

- Si on prend *i* = 4, alors la première chaine considérée est "POMM".
- Si on prend *j* = 3, alors la seconde chaine considérée est "PAU".

Visualisation de *i* et *j*

Pour visualiser les deux variables, utilisons un tableau : *i* = 4 est encadré en violet (pour toute valeur de *j*), et *j* = 3 est encadré en orange (pour toute valeur de *i*).

L'intersection des deux donne la distance de Levenshtein entre "POMM" et "PAU", soit :

	i (0)	1	2	3	4	5
j (0)	----------	P	O	M	M	E
1	P					
2	A					
3	U				$lev_{a,b}(4;3)$	
4	M					
5	E					

$lev_{a,b}(4;3)$

Comment remplir la matrice ? (Condition préliminaire)

Pour remplir la matrice, il faut remplir toutes les cases, soit calculer toutes les sous-distances de Levenshtein.

On définit alors la variable *C* : Coût de l'opération (7)

Il faut néanmoins prendre en compte une condition préliminaire :

Pour $0 \leq i \leq |a|$, pour $0 \leq j \leq |b|$,

Si $x_i \leftrightarrow \mathrm{y}_j$, alors $C = 0$.

(Si un ensemble de lettres a la même position dans les deux mots, alors leur coût vaut 0. On ne va donc pas modifier cet ensemble étant donné qu'il est identique en terme de position dans le mot tapé et le mot du dictionnaire, et on ne le compte donc pas dans la distance de Levenshtein.)

Exemple de la condition préliminaire

Exemple : Chaine a : POMME Chaine b : PAUME

Nous pouvons ici retrouver les mêmes lettres aux positions 1, 4 et 5 dans les deux mots.

Comme leurs positions sont identiques, il est inutile de les prendre en compte dans le calcul de distance. Il suffit alors de calculer la distance de Levenshtein entre **les deux chaines qui restent,** d'où :

$$lev_{a,b}(POMME, PAUME) = lev_{a,b}(OM, AU) = 2$$

Comment remplir la matrice?

Si les deux chaines **ne remplissent pas la condition**, on définit alors la distance de Levenshtein entre les deux chaines de cette manière (avec $C = 1$ strictement, sauf pour la substitution):

$$lev_{a,b}(i;j) = \min \begin{cases} d(i-1;j)+1 & Suppression \\ d(i;j-1)+1 & Insertion \\ d(i-1;j-1)+C & Substitution \end{cases}$$

Par exemple, calculons $lev_{a,b}(1;1)$ pour la chaine "POMME" et la chaine "PAUME".

On obtient alors, pour $i = 1$ et $j = 1$:

$$lev_{a,b}(1;1) = \min \begin{cases} d(0;1)+1 & = 1+1 = 2 \\ d(1;0)+1 & = 1+1 = 2 \\ d(0;0)+C & = 0+1-1 (C = 0 \ car \ i \leftrightarrow j \end{cases}$$

Ici, la valeur minimale est 0. Donc, $lev_{a,b}(1;1)$ = 0. On met cette valeur dans la matrice, et ainsi de suite.

(8)

Visualisation par une matrice

En suivant cette méthode de calcul, on peut aisément remplir les premières cases :

- lev(P,P) = 0 (le **même caractère à la même position)**
- lev(P,PA) = **1 (insertion** d'un A)
- lev(PM,P) = **1 (insertion** d'un M)

On obtient alors directement :

----------	P	M	O	M	E
P	0	1			
A	1				
U					
M					
E					

Visualisation finale par la matrice

Si on continue de remplir la matrice en utilisant la distance minimale, on obtient la matrice finale suivante.

j/i	0	1	2	3	4	5
0	----------	P	M	O	M	E
1	P	0	1	2	3	4
2	A	1	1	2	3	4
3	U	2	2	2	2	4
4	M	3	2	3	2	3
5	E	4	3	3	3	2

En remplissant toutes les cases du tableau, on obtient finalement toutes les sous-distances entre toutes les chaines de caractères.

Dans la case tout en bas à droite, on obtient la distance de Levenshtein entre les chaines telles que i = 5 et *j* = 5, soit la distance de Levenshtein entre les deux chaines principales.

Le nombre dans la case est 2, on en conclut alors que la distance de Levenshtein entre "PMOME" et "PAUME" vaut 2, ce qui équivaut à ce que nous avions trouvé lors des simulations.

On retrouve alors bien une distance de Levenshtein de 2, comme vu dans l'exemple. (9)

Conclusion

- Les mots POMME et PAUME ont la distance de Levenshtein la plus faible par rapport au mot tapé, et semblent donc les plus cohérents à suggérer en priorité.
- POMMIER est donc le moins cohérent, et ne sera pas suggéré.
- La distance de Levenshtein s'applique à n'importe quel mot, et peut être déterminée tant par des simulations et des essais, que par une matrice.

Mot tapé par l'utilisateur	P	M	O	M	E
Mots les plus cohérents	P	O	M	M	E
	P	A	U	M	E

7. Méthode du clavier

Introduction de la méthode

Après avoir repris les calculs de similitude effectués précédemment, une autre approche est envisageable en fonction du clavier utilisé pour suggérer des mots. Nous allons pour cela décomposer les mots en tranches d'une lettre seulement.
Il existe différents types de claviers, tels que QWERTY, QWERTZ ou encore AZERTY. Nous allons réaliser nos exemples avec les claviers de type QWERTZ.

Explication de la méthode

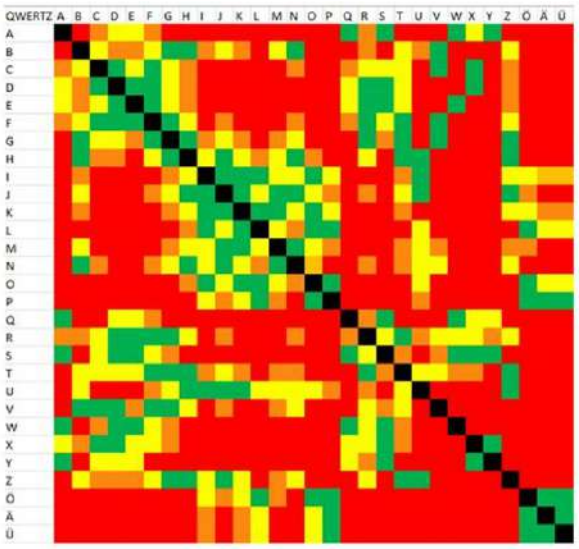
Nous avons remarqué que chaque touche sur un clavier est entourée de 2 à 6 lettres. Pour la touche en question, toutes les autres touches qui l'entourent sont à ce que nous appelons une unité de distance. Nous considérons toutes les touches à une unité de distance comme la 'marge' d'erreur.
Nous commençons en faisant des essais individuels, puis 2 à 2, 3 à 3, 4 à 4 etc.
Par exemple, on considère que le mot tapé par l'utilisateur est : ''qrtistique''. Le programme va rechercher le nombre d'unités de distance de chaque lettre par rapport à Q dans la base. Il va ensuite essayer différents remplacements (en singletons, en binômes, en trinômes...) Cette méthode va être testée pour chaque lettre. Enfin, il affichera tous les mots existants cohérents trouvés.

Bases de données

Le premier cercle, ici le vert, aura pour diamètre 2 touches. Par définition, les cercles ont un diamètre de 2n touches où n est le nombre du cercle, pour le cercle vert on aura donc n = 1. Pour le cercle qui suit, le jaune, le diamètre sera de 2n touches et n sera alors égal à 2. On répètera cette opération autant de fois que nécessaire pour obtenir le nombre de cercles voulu.
Suivant les cercles, nous pouvons rapprocher les lettres les plus proches entre elles sur le clavier et changer les lettres suivant leur position.

Légende:

En vert nous aurons les lettres à 1 unité de Distance
En jaune les lettres à 2 unités de distance
En orange les lettres à 3 unités de distance
En rouge les lettres à plus de 3 unités de distance



Cette base de données représente un tableau à double entrée avec les cases présentant la distance entre les lettres dans un clavier QWERTZ.

Questions à suivre pour cette méthode

Par manque de temps, nous nous sommes posé différentes questions pour pouvoir poursuivre cette méthode :

- Que fait-on du tableau par la suite ?
- Que fait-on avec les distances ---> Que fait-on si un mot est modifié par une lettre à côté de la lettre fausse dans le clavier ?
- Comment dire à l'utilisateur que le mot est maintenant corrigé ?

Conclusion

Pour répondre à la problématique, nous avons donc trouvé trois méthodes : la méthode des combinaisons, la distance de Levenshtein et la méthode du clavier.

Durant cette année MATh.en.JEANS, nous avons appris :

- à réfléchir à un problème et de rechercher à comment le résoudre
- à coder sur Python
- à développer des organigrammes
- à apprendre de nos erreurs

Notes d'édition

(1) Il aurait été intéressant d'expliquer la méthode utilisée par ce programme pour déterminer les mots similaires. Par ailleurs, le lien ne fonctionne pas.

(2) Il faudrait ici préciser ce qu'on entend par « lettres similaires » et « nombre de lettres total » : de quel mot ? dictionnaire ou tapé ?

(3) Il est pertinent de noter qu'il y a plusieurs combinaisons pour un nombre impair de lettres, mais on n'explique pas ensuite comment choisir le découpage.

(4) Le lien donné conduit à un organigramme intéressant mais il aurait fallu expliquer les grandes lignes de cet organigramme.

(5) Ce lien conduit à un accès refusé.

(6) Il faudrait justifier que le nombre **minimal** d'opérations est bien 4 pour passer de POMME à POMMIER.

(7) La variable C n'est pas clairement définie, ni utilisée dans la suite.

(8) Attention aux notations : $d(i;\ j)$ n'est pas défini.

(9) Il manque des exemples qui montrent l'utilité de l'introduction de cette matrice : sur des dimensions plus grandes par exemple ou pour un traitement informatique…