

# ALGORITHMIQUE DU TEXTE

## RAPPORT DU DEVOIR MAISON 2020

Objectif :

Réalisation d'un correcteur orthographique

Réalisé par:

KHEMIM Oussama - 21915053  
OUANGUICH Mouad - 21813909

## Question 1

La distance Damerau-Levenshtein qui sépare les deux mots “faute\_de\_frappe” et “fote\_de\_frappe” est égale à 3.

faute\_de\_frappe (suppression) -> fute\_de\_frappe (substitution) -> fote\_de\_frappe (transposition) -> fote\_de\_farppe

## Question 2

```
Entrée [4]: def distDL(u, v):
    # "Infini" -- Utilisées pour empêcher les transpositions des premiers caractères
    INF = len(u) + len(v)

    # Matrix: (M + 2) x (N + 2)
    matrix = [[INF for n in xrange(len(v) + 2)]]
    matrix += [[INF] + range(len(v) + 1)]
    matrix += [[INF, m] + [0] * len(v) for m in xrange(1, len(u) + 1)]

    # Contient la dernière ligne de chaque élément rencontré: DA dans le pseudocode Wikipedia
    last_row = {}

    # remplissage de la table des couts
    for row in xrange(1, len(u) + 1):
        # Caractères courant dans u
        ch_u = u[row-1]

        # la colonne du dernier match de la ligne: DB dans le pseudocode
        last_match_col = 0

        for col in xrange(1, len(v) + 1):
            # Caractères courant dans v
            ch_v = v[col-1]

            # Dernière ligne avec un match
            last_matching_row = last_row.get(ch_v, 0)

            # Cout de la substitution
            cost = 0 if ch_u == ch_v else 1

            # Calcul de la distance de la sous-chaine
            matrix[row+1][col+1] = min(
                matrix[row][col] + cost, # Substitution
                matrix[row+1][col] + 1, # Addition
                matrix[row][col+1] + 1, # Deletion

                # Transposition
                matrix[last_matching_row][last_match_col]
                # Cout des lettres entre celles transposées
                # 1 addition + 1 suppression = 1 substitution
                + max((row - last_matching_row - 1),
                    (col - last_match_col - 1))
                # Cout de la transposition
                + 1)

            # S'il y a match, on met last_match_col à jour
            if cost == 0:
                last_match_col = col

        # Mettre à jour la dernière ligne pour le caractère courant
        last_row[ch_u] = row

    # Retourner le dernier élément
    return matrix[-1][-1]
```

### TESTS:

La distance entre les deux mots “faute\_de\_frappe” et “fote\_de\_farppe” est 3.

```
[84]: distDL("faute_de_frappe", "fote_de_farppe")  
]: 3
```

La distance entre le mot “marine” et le mot “mouad” est égale à 5.

```
Entrée [85]: distDL("marine", "mouad")  
Out[85]: 5
```

### Question 3

On charge notre dictionnaire à l’aide de la fonction `dict_words(dict_file)`

```
def dict_words(dict_file):  
    with open(dict_file, 'r') as file:  
        data = file.read()  
    return data.split()
```

```
dict = dict_words("list.txt")  
  
def calcul_distance_dict(dict, mot):  
    for word in dict:  
        print("distance("),  
        print(word),  
        print(", "),  
        print(mot),  
        print(") ="),  
        print(distDL(word, mot))
```

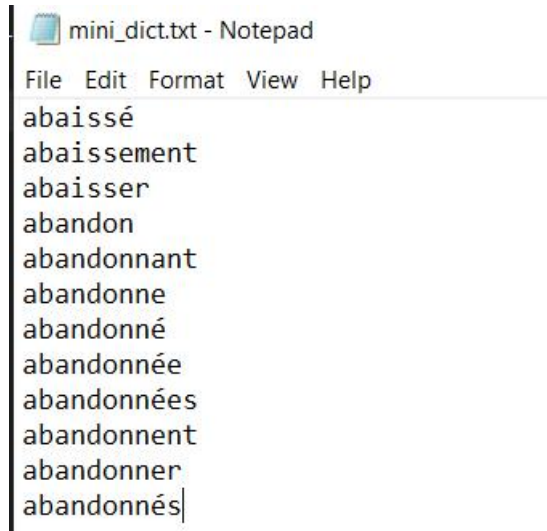
### TESTS:

Pour le mot “agua” et le dictionnaire suivant : [algo, tango, alto, lagon, algue, brio, logo]. La distance entre le mot “agua” et tous les mots du dictionnaire est :

```
calcul_distance_dict(dict, "agua")
```

```
distance( algo , agua ) = 3  
distance( tango , agua ) = 4  
distance( alto , agua ) = 3  
distance( lagon , agua ) = 3  
distance( algue , agua ) = 2  
distance( brio , agua ) = 4  
distance( logo , agua ) = 4
```

- Pour le mot "*bias*" et le dictionnaire suivant :



```
mini_dict.txt - Notepad  
File Edit Format View Help  
abaissé  
abaissement  
abaisser  
abandon  
abandonnant  
abandonne  
abandonné  
abandonnée  
abandonnées  
abandonnent  
abandonner  
abandonnés
```

Output:

```
distance( abaissé , bias ) = 5  
distance( abaissement , bias ) = 8  
distance( abaisser , bias ) = 5  
distance( abandon , bias ) = 6  
distance( abandonnant , bias ) = 9  
distance( abandonne , bias ) = 8  
distance( abandonné , bias ) = 9  
distance( abandonnée , bias ) = 10  
distance( abandonnées , bias ) = 10  
distance( abandonnent , bias ) = 10  
distance( abandonner , bias ) = 9  
distance( abandonnés , bias ) = 9
```

Après modification du programme pour afficher tous les mots du dictionnaire à distance au plus e de mot

Le test est fait pour le mot "agua" et le dictionnaire [algo, tango, alto, lagon, algue, brio, logo] avec e = 3.

```
[93]: dict = dict_words("list.txt")

def calcul_distance_dict(dict, mot, e):
    for word in dict:
        dist = distDL(word, mot)
        if dist <= e:
            print("distance(",
                  print(word),
                  print(", "),
                  print(mot),
                  print(") ="),
                  print(dist)

calcul_distance_dict(dict, "agua", 3)
```

```
distance( algo , agua ) = 3
distance( alto , agua ) = 3
distance( lagon , agua ) = 3
distance( algue , agua ) = 2
```

#### Question 4:

```
def time_of(func, *args):
    import time
    t = time.time()
    res = func(*args)
    print("Time: ", (time.time() - t))
    return res
```

La fonction "time\_of" prend en paramètre le nom de la fonction à tester et ses paramètres.

Le temps d'exécution obtenu pour des dictionnaires de petites tailles est négligeable donc on va essayer sur un dictionnaire français de 23066 mots distincts.

Pour le mot "accommodate" de longueur 11

Les résultats d'exécution pour e = 5

```
print(time_of(calcul_distance_dict, dict, "accommodate", 5))
```

Résultat : ('Time: ', 2.2190001010894775)

Les résultats de l'exécution avec  $e = 11$  :

```
print(time_of(calcul_distance_dict, dict, "accommodate", 11))
```

Resultat : ('Time: ', 44.634000062942505)

Le programme traite 23066 mots en calculant la distance à chaque fois avec un mot de longueur 11 en moins de 10 secondes. Le temps d'exécution est par conséquent satisfaisant.

En rajoutant le paramètre erreur, le temps est plus conséquent surtout avec une marge d'erreur plus ou moins grande.

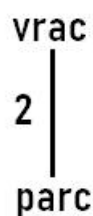
### Question 5:

La complexité en temps de notre algorithme est  $O(m*n)$ . La raison est l'espace et non le temps parce que l'algorithme construit une matrice de taille  $n*m$ .

### Question 6:

"vrac" est le mot à la racine de l'arbre.

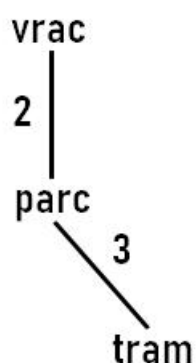
Le deuxième mot est "parc" ayant une distance de 2 de "vrac", l'arbre à la première étape ressemble à ceci :



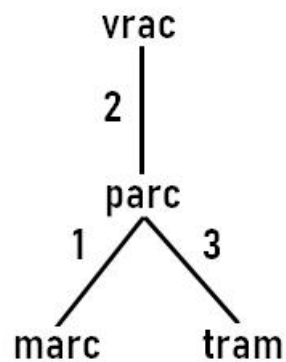
La distance entre le mot à la racine et le mot suivant ("tram") est 2 qui correspond au poids de la première arête créée.

La distance à calculer dans ce cas est celle de "parc" et "tram" qui est égale à 3.

L'arbre par conséquent ressemble à ceci :

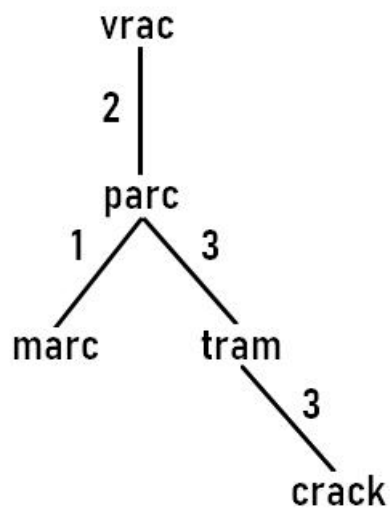


Le mot “marc” est d’une distance de 2 avec “vrac” et de 1 avec “parc”.  
On ajoute l’arête de poids 1 entre “parc” et “marc”.



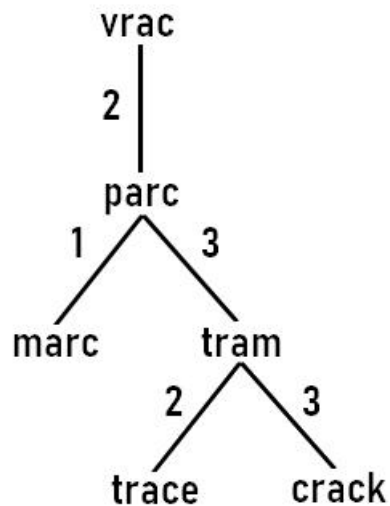
La distance entre “crack” et “vrac” est 2; 3 avec “parc” et elle est égale à 3 avec “tram”.

Une arête de poids 3 est donc créée entre “tram” et “crack” comme ceci :



Le dernier mot “trace” est d’une distance successivement de 2, 3 et 2 avec “vrac”, “parc” et “tram”.

L’arbre de Burkhard-Keller pour la liste [*vrac, parc, tram, marc, crack, trace*] est le suivant :



### Question 7:

```

def construireArbBK(dict):
    it = iter(dict)
    racine = next(it)
    arbBK = (racine, {})

    for i in it:
        inserer_mot(arbBK, i)
    return arbBK

def inserer_mot(arbBK, mot):
    """
    Ajoute le mot à arbBK
    :param mot: le mot à insérer
    """
    noeud = arbBK
    if noeud is None:
        arbBK = (mot, {})
        return

    while True:
        parent, fils = noeud
        distance = distDL(mot, parent)
        noeud = fils.get(str(distance))
        if noeud is None:
            fils[str(distance)] = (mot, {})
            break
  
```

### TESTS:

Le dictionnaire est la liste [algo, tango, alto, lagon, algue, brio, logo]



```
Entrée [19]: tree = construireArbBK(dict)
```

```
Entrée [20]: tree
```

Output:

```
('algo',  
 {'1': ('alto', {}),  
  '2': ('tango', {'3': ('lagon', {'2': ('logo', {})}), '4': ('algue', {})}),  
  '3': ('brio', {}))
```

Le temps d'exécution de cet algorithme sur un dictionnaire de cette taille est négligeable.

Voyons sur un dictionnaire de taille considérable.

Le dictionnaire utilisé contient 23066 mots français distincts.

```
5]: print(time_of(construireArbBK, dict))
```

```
('Time: ', 11.253000020980835)  
(('abaiss\xe9', {'11': ('accessoirement', {'11': ('administrateur', {'1  
1': ('antiterroriste', {'11': ('bureaucratie', {'11': ('constitution',  
{ '12': ('d\xe9vergonder', {}), '7': ('inscriptions', {})}), '10': ('co  
nfirmation', {'9': ('signifierait', {}), '8': ('consid\xe9rables', {'8  
' : ('contrecarrer', {})}), '12': ('jurisprudence', {}), '10': ('docume  
ntaires', {'10': ('hippopotame', {'8': ('morphinomane', {})}), '7': ('  
gestionnaire', {})}), '12': ('insignifiant', {}), '9': ('documentair  
e', {'9': ('pr\xe9senterait', {}), '8': ('rencontrera', {}), '12': ('t  
ranscription', {}), '10': ('restrictives', {})}), '10': ('bicentenai  
re', {'9': ('contraignante', {}), '10': ('congressistes', {'9': ('Mont  
fermeil', {}), '11': ('productrice', {})}), '11': ('progressistes',  
{}, '7': ('scientifique', {}), '8': ('\xe9gocentrique', {'9': ('inten  
tionnel', {}), '10': ('sentimentale', {})}), '13': ('consultation',  
{ '11': ('respectueux', {}), '12': ('deutschemark', {})}), '12': ('conc  
iliation', {'11': ('essentiell', {'9': ('repr\xe9sentera', {}), '10  
' : ('passe-partout', {'10': ('surprenantes', {})}), '10': ('enguirla  
nder', {'9': ('s\xe9lectionner', {})}), '12': ('surench\xe8res', {}),  
'4': ('conformation', {'5': ('consignations', {})}), '7': ('concordant  
...'))
```

Question 8:

### Question 9:

```
def chercherArbBK(arbBK, mot, e):  
  
    def recuperer_liste(parent):  
        p_mot, fils = parent  
        distance = distDL(mot, p_mot)  
        res = []  
        if distance <= e:  
            res.append((distance, p_mot))  
  
        for i in range(distance - e, distance + e + 1):  
            f = fils.get(str(i))  
            if f is not None:  
                res.extend(recuperer_liste(f))  
        return res  
  
    # tri ascendant par distance  
    return sorted(recuperer_liste(arbBK))
```

### TESTS:

#### Soit:

Le dictionnaire est la liste [algo, tango, alto, lagon, algue, brio, logo]

Le mot recherché est : "bias".

L'erreur e = 4.

La distance entre le mot "bias" et tous les mots du dictionnaire est :

```
[48]: dict = dict_words("list.txt")  
for mot in dict:  
    print("la distance entre"),  
    print(mot),  
    print("et le mot bias est :"),  
    print(distDL(mot, "bias"))  
  
la distance entre algo et le mot bias est : 4  
la distance entre tango et le mot bias est : 5  
la distance entre alto et le mot bias est : 4  
la distance entre lagon et le mot bias est : 5  
la distance entre algue et le mot bias est : 5  
la distance entre brio et le mot bias est : 3  
la distance entre logo et le mot bias est : 4
```

Le résultat de la recherche est le suivant :

```
[7]: chercherArbBK(tree, "bias", 4)  
[(3, 'brio'), (4, 'algo'), (4, 'alto'), (4, 'logo')]
```

Résultat du temps d'exécution sur le grand dictionnaire avec :

- mot recherché = "bias" et e = 4

```
2]: print(time_of(chercherArbBK, tree, "bias", 4))
('Time: ', 0.5550000667572021)
```

- mot recherché = "accommodation" et e = 8

```
4]: print(time_of(chercherArbBK, tree, "accommodation", 8))
('Time: ', 2.0840001106262207)
```

- mot recherché = "anticonstitutionnel" et e = 20

```
6]: print(time_of(chercherArbBK, tree, "anticonstitutionnel", 20))
('Time: ', 3.7689998149871826)
```

### Question 10:

```
def sauvegarderArbBK(arbBK, chemin):
    import json
    if not chemin:
        import os
        chemin = os.path.join(os.getcwd(), 'tree.json')
    with open(chemin, 'w') as file:
        file.write(json.dumps(arbBK))

def restaurerArbBK(chemin):
    import json
    if not chemin:
        import os
        chemin = os.path.join(os.getcwd(), 'tree.json')
    with open(chemin, 'r') as file:
        arbBK = json.loads(file.read())
    return arbBK
```

### TESTS:

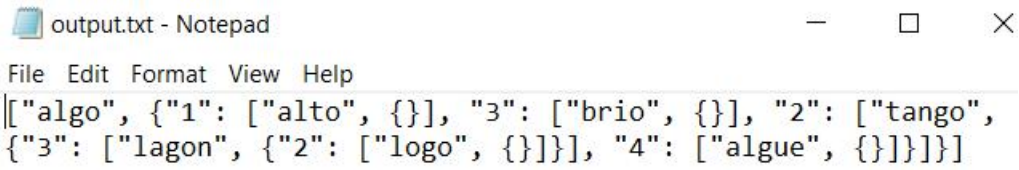
### SAUVEGARDE:

Pour l'arbre construit à partir du dictionnaire [algo, tango, alto, lagon, algue, brio, logo]

La sauvegarde de l'arbre donne le résultat suivant :

```
sauvegarderArbBK(tree, "output.txt")
```

Output:



```
output.txt - Notepad
File Edit Format View Help
[["algo", {"1": ["alto", {}], "3": ["brio", {}], "2": ["tango",
{"3": ["lagon", {"2": ["logo", {}]}], "4": ["algue", {}]}]]]
```

### RESTAURATION:

La restauration de l'arbre contenu dans le fichier "output.txt" :

```
[82]: arbreBK = restaurerArbBK("output.txt")
      print(arbreBK)

[ 'algo', { '1': [ 'alto', {}], '3': [ 'brio', {}], '2': [ 'tango', { '
3': [ 'lagon', { '2': [ 'logo', {}]}], '4': [ 'algue', {}]}]]]
```