# Travail sur la compléxité : Anagrammes

La compléxité algorithmique est une notion assez abstraite en informatique.

*Rappel : La complexité d'un algorithme correspond au nombre d'opérations* ***élémentaires*** *réalisées au court d'un algorithme* ***en fonction de la taille de la donnée utilisée****.*
*La compléxité d'un algorithme peut être associé à une fonction de complexité:*

***constante, linéaire, semi-linéaire, logarithmique, quadratique, exponentielle***.

*Exemple : pour le tri par insertion d'une liste de taille $n$, la complexité de l'algorithme correspond environ au nombre d'échange de valeurs et du nombre de parcours de la liste qui sont répétés au maximum $n^2$. On écrit que $C_{tri\ par\ insertion} = O(n^2)$.*

Cette activité permettra de mettre en valeur que l'implantation d'un algorithme peut se faire de différentes manières et que la complexité de celui-ci peut être totalement différentes.

Un anagramme d'un mot est un mot qui peut s'écrire avec les mêmes lettres qu'un autre.
Par exemple : **Typhon** est l'anagramme de **Python**.

Ce TP permettra de vous sensibiliser à ce qu'est la complexité algorithmique et quelle est son importance grâce à une étude de courbe de complexité.

*Vous écrirez l'intégralité des réponses aux questions dans le fichier Python fourni*

## Partie 1 : Anagramme grâce à une clef

*Le but de cette première partie est de trouver les anagrammes d'un mot passé en paramètre d'une fonction anagramme et qui parcourra tout le lexique de mot fourni pour trouver ses anagrammes.*

Un mot peut être associé à une clef. Cette clef, pour nous simplifier la tâche peut correspondre aux lettres en minuscule, sans accent, triées par ordre croissant.

- Question 1 : **Ecrire un dictionnaire *equivalences* qui permet d'associer à tous les caractères avec accent, leur équivalent sans accent.**

```
>>> print(equivalence['é'])
e
```

- Question 2 : **Ecrire une fonction python sans_accent qui prend en paramètre une mot en minuscule et renvoie son écriture sans accent.**

```
>>> print(sans_accent('orangé'))
orange
```

On souhaite trier la chaine de caractère par ordre alphabétique pour trouver des anagrammes correspondant plus facilement. En effet, deux anagrammes auront les mêmes lettres en même nombre triées dans l'ordre alphabétique, on aura donc notre clef.
Le soucis est que la méthode `sort` en Python n'existe pas pour les chaines de caractères. On va donc la créer nous-même.

- Question 3 : **Compléter la fonction *clef_mot* suivante pour qu'elle se comporte comme la docstring l'indique**

```
def clef_mot(chaine:str)->str:
    """
    Cette méthode transforme la chaine de caractère passée en paramètre en liste.
    Elle la trie en utilisant la méthode sort des listes Python.
    Pour chaque caractère de la liste :
        Elle ajoute la caractère dans une chaine de caractère qui sera renvoyée.
    """
    s = ""
    l = list(...)
    l.sort()
    for ... :
        s = s + ...
    return s
```

On sait donc que deux mots qui ont la même clef sont des anagrammes.

- Question 4 : **En déduire une fonction `anagramme_via_clef` à l'aide des précédentes, qui prend en paramètre deux chaines de caractère et renvoie *True* si les deux caractères sont des anagrammes, *False* sinon.**

On veut réaliser l'étude de la complexité sur un Lexique conséquent.
On rappelle que la complexité dépend de la **taille de l'entrée**.
Pour ce faire, on va supposer que l'on traite la complexité temporelle qui est facilement mesurable en python.
Grâce au module **timeit** on peut chronométrer des actions.
On va donc mesurer le temps nécessaire pour trouver des anagrammes pour des lexiques de plus en plus grands et les reporter sur une courbe. Cette courbe sera tracée grâce au module **matplotlib**.

# Complexité de l'algorithme

La fonction suivante `mesure_anagramme_clef` correspond à la recherche du temps necessaire pour trouver les anagrammes d'un nombre de mots.

```
def mesure_anagramme_clef(nb_mots):
    '''
    Réalise la mesure du temps necessaire pour trouver
    les anagrammes de i mots sur un lexique d'une taille i.

    La méthode employée correspond à la création d'une clef
    et de la recherche des clefs des autres mots auquel on le compare.

    La complexité dépend de la taille du Lexique.
    '''
    tab_mesures =[]

    for i in range(0,nb_mots):

        # Temps du début
        temps_debut = timeit.default_timer()
        for mot in LEXIQUE[:i]:
            anagramme_1(mot, LEXIQUE[:i])

        # Temps de fin
        temps_fin = timeit.default_timer()
        # Ajout de la durée dans le tableau : la case i correspond à la
        # recherche des anagrammes de i mots sur un lexique de i mots
        tab_mesures.append(temps_fin-temps_debut)

    print(str(sum(tab_mesures)) + " secondes")

    # Création d'un graphique : l'abscisse correspond au nombre de mot
    # et l'ordonnée correspond au temps necessaire
    plt.plot([i for i in range(nb_mots)], tab_mesures)
    # Création d'un fichier png correspondant à la courbe créée grâce à la méthode plot.
    plt.savefig("fig_clef.png")
    plt.clf()
```

Question 5 : **Utiliser la fonction mesure_anagramme_clef sur une liste de 500 mots. Combien de temps cela a pris ?**

Question 6 : **Ouvrez le fichier `fig_clef.png`. Quelle est la forme de la courbe? À quelle complexité cela correspond ?**

# Partie 2 : Anagramme grâce à un dictionnaire

Le problème de la méthode précédente est que l'on parcourt le lexique pour chaque mot et on cherche dans cette liste tous les anagrammes.
Cela veut dire que pour chaque élément on parcourt chaque élément alors que cela pourrait se faire en un seul passage dans le lexique.

Une structure de données que l'on peut employer pour trouver en un coût unitaire serait l'emploi d'un dictionnaire.
En effet, on peut parcourir une seule fois la liste, remplir petit à petit le dictionnaire qui prend pour clef les clefs des mots obtenues à l'aide de la fonction `clef_mot` réalisée précédemment et en valeurs les mots associés.
L'accès à une valeur d'un dictionnaire se fera en un coût de un.

Question 7 : **Créer une fonction `dictionnaire_du_lexique` qui prend en paramètre un lexique et renvoie le dictionnaire associé.**

Question 8 : **En déduire une fonction `anagramme_via_dictionnaire` qui prend en paramètre un mot et un dictionnaire et renvoie la liste des anagrammes d'un mot d'un lexique.**

Question 9 : **En vous inspirant de la fonction `mesure_anagramme_clef`, en déduire une fonction `mesure_anagrame_dico` qui correspond à la mesure du temps necessaire pour trouver les anagrammes d'un mot.** Cette fonction utilisera la méthode de recherche d'anagramme via un dictionnaire. Celle-ci utilisera les fonctions créées précédemment et mettra le graphique créé dans un fichier `fig_dico.png`

Question 10 : **Regarder le fichier `fig_dico.png`, à quelle fonction la courbe peut être associée? À quelle complexité cela se rapporte? Cela est-il meilleur qu'aux premières mesures?**

Question 11 : **Quelle conclusion peut-on en déduire entre l'implémantation d'un algorithme et sa complexité?**