

# 1.Comptage base-à-base

L'estimation de la distance base-à-base ne dépend pas du type de mutation car, si une autre mutation que la mutation de remplacement a lieu, la différence sera un caractères différents ou une chaîne avec moins de caractères.

Ces différences étant dore et déjà prises en compte par la méthode, la réponse sera donc la même en présence d'autres mutations.

[exemple avec que remplacement]

[exemple avec autre mutation]

## 2.Distance de Levenshtein

```
# dernière ligne a jour du tableau de levenshtein
distance_row: list[list[int]] = list(range(compared_sequence_length + 1))

for row, initial_char in enumerate(initial_sequence):

    current_row = [row + 1] + [0]*compared_sequence_length

    for column, compared_char in enumerate(compared_sequence):

        # coût d'action a 1
        cost: int = 1
        corner_distance = distance_row[column + 1]
        up_line_distance = current_row[column]
        left_column_distance = distance_row[column]

        if initial_char == compared_char:
            cost = 0 # pas besoin de changement les valeurs

        distance: int = min(corner_distance, up_line_distance, left_column_distance) + cost
        current_row[column + 1] = distance

    distance_row = current_row

return distance_row[-1]
```

L'algorithme de Levenshtein consiste en 2 boucles imbriquées avec un nombre d'itération équivalent à la longueur des 2 chaînes de caractères données.

L'ensemble des autres instructions sont d'ordre  $O(1)$ , de même pour les accès aux valeurs de la liste.

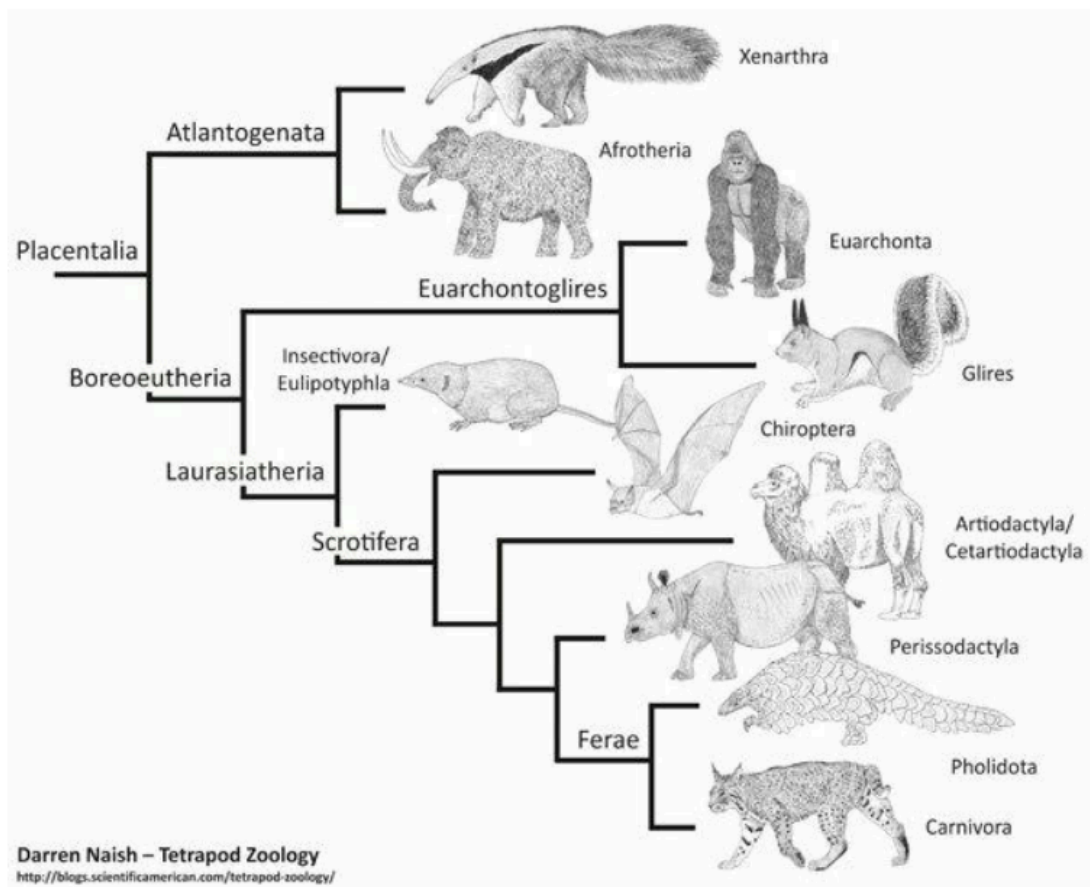
Par conséquent, cette fonction est d'ordre  $O(k \cdot p)$ , avec  $k$  la longueur de la première chaîne de caractère,  $p$  étant la longueur de la seconde.

Dans un cadre plus général, nous pouvons argumenter que les 2 chaînes de caractères sont de longueur similaire, par conséquent l'ordre temporel de cette fonction est  $O(n^2)$ .

Pour ce qui est de la mémoire, nous avons pu optimiser l'algorithme en se basant sur le travail présent sur le repository github de [TheAlgorithms](https://github.com/TheAlgorithms).

En suivant l'idée qui nous avait traversé l'esprit, lors du début de la seconde partie du projet, nous avons, au lieu de garder en mémoire un tableau dont la majorité des valeurs ne sont utilisées que très rarement et deviennent vite obsolètes, nous ne gardons en mémoire que la ligne complète précédente et la ligne en cours de création, l'utilisation mémoire est alors de  $O(2n)$  avec  $n$  la taille du mot le plus long.

### 3. Arbres phylogénétiques



#### Question 8 :

Le dernier ancêtre commun du gorille et du chameau est l'ancêtre hypothétique Boreoeutheria. Pour l'ancêtre commun de la chauve-souris et lynx l'ancêtre commun est Laurasiatheria

## Question 9:

Parmi les distances suivantes `distance(carnivora, pholidota)`, `distance(carnivora, artiodactyla)` et `distance(carnivora, chiroptera)`. La distance entre `carnivora` et `pholidota` est la plus courte parmi les 3 distances étant donné qu'ils sont des enfants de leur ancêtre commun qui est `Ferae`. Ensuite, entre `Carnivora` et `artiodactyla` la distance est plus grande que pour le cas précédent et il faut remonter à `Costifera` pour avoir leur ancêtre commun.

La méthode **`mutation_replacement_distance()`** permet de comparer deux séquences et de connaître la distance. Ainsi une séquence peut avoir des bases différentes mais également des longueurs différentes. La méthode vérifie s'il y a une différence de longueur entre les deux fichiers et ajoute la différence à la somme. Puis compte les différences sur la séquence restante sans prendre la partie du texte qui est plus long.

Méthodes de lecture et écriture de séquence :

La méthode de lecture **`sample_reader()`** nous permet de récupérer la séquence ADN d'un fichier, elle permet également de supprimer les commentaires afin d'avoir uniquement les Nucléotides afin de pouvoir utiliser les différents algorithmes que nous avons implémentés. Pour réaliser cette méthode de lecture nous utilisons la méthode `open` qui permet de lire un fichier. Nous lisons ce fichier uniquement par bloc de 5000 caractères afin de limiter l'usage de mémoire en cas de très gros fichiers de séquences ADN. Ensuite nous utilisons une liste de compréhension afin de supprimer les commentaires et de n'avoir plus que les nucléotides. On met ensuite cette liste dans un `String` et j'ajoute ce `string` dans une liste de bloc nettoyés ce qui permet d'éviter de faire des concaténations successives de `String` ce qui n'est pas très efficace. Enfin, une fois le fichier lu en entier, la méthode ajoute tous les blocs à une chaîne de caractères grâce à la méthode `join`.

En ce qui concerne la lecture, la méthode **`sample_writer()`** utilise également `open`. Le paramètre `w` est utilisé puisqu'il permet d'écraser le fichier si il existe déjà et qu'il est modifié comme lors d'une mutation ou si il n'existe pas il est créé.

## Question 10:

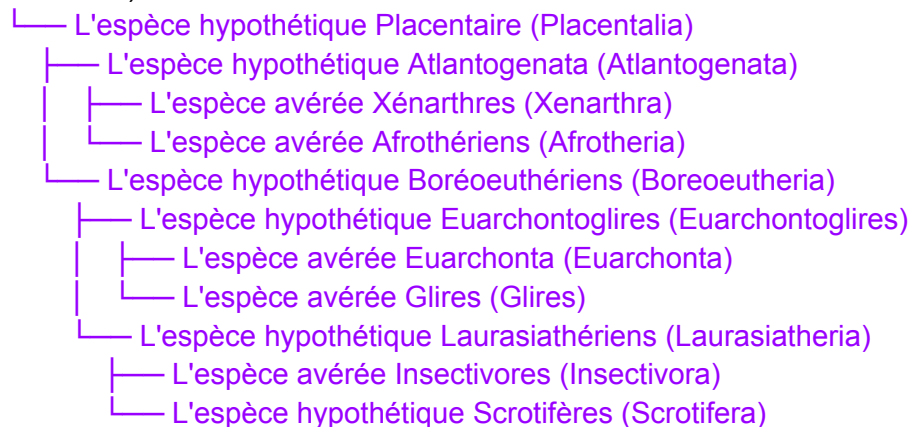
L'implémentation des 2 classes **Espèce Hypothétique** et **Espèce Avérée** permettent de différencier les espèces qui sont des ancêtres communs à plusieurs espèces (**Espèce Hypothétique**), de ces espèces non hypothétiques (**Espèce Avérée**).

J'ai procédé à une implémentation avec l'espèce hypothétique avec comme attributs: son nom, son nom scientifique et ses enfants. L'espèce hypothétique a comme méthodes, des `getter` et des `setters`, mais aussi des méthodes pour ajouter et supprimer des enfants de la liste des enfants.

L'espèce avérée est un enfant de la classe **Espèce Hypothétique**, elle va donc récupérer les attributs et les méthodes de la classe parent, la différence avec la classe **Espèce**

Hypothétique est que l'Espèce Avérée possède comme attribut "genome" qui va être une séquence ADN qui la représente.

Cet implémentation est efficace par son nombre minimal de classe et de l'utilisation de la généralisation pour éviter de faire des doublons et de rendre le programme moins performant. De plus, la méthode `__str__()` suis une logique d'arbre fait dans la méthode `_build tree string()` qui construit un arbre sous cette forme (en prenant l'exemple de la figure ci-dessus):



## Question 11:

Avec une implémentation de la méthode **calculer\_distance()**, on peut permettre de calculer la distance entre les espèces hypothétiques, mais aussi entre une espèce hypothétique et une espèce avérée, sachant que le calcul de distance se fait à partir des génomes.

On pourra calculer les distances des espèces hypothétiques en se servant des distances entre les génomes, cela peut être calculer grâce à la méthode de la question 9 :

**mutation\_replacement\_distance()**.

L'idée initial du calcul de distance était de se servir de la distance de hamming avec la méthode **distance\_hamming()**, le problème étant qu'elle n'était pas assez robuste et qu'elle était trop simple par rapport à **mutation\_replacement\_distance()** car elle ne couvrait pas toute les différences pour calculer sa distance.

La méthode **calculer\_distance()** gère si les 2 espèces sont pareil et gère récursivement les espèces qui sont hypothétique qu'ils aient des enfants ou non (si les deux espèces n'ont pas d'enfant, on retourne 1)

```

def calculer_distance(espece1, espece2):
    """
    Calcule la distance entre deux espèces, qu'elles soient avérées ou hypothétiques.

    Param espece1 : Une espèce
    Param espece2 : Une autre espèce

    Return : la distance entre espece1 et espece2
    """
    # Dans le cas où les 2 espèces sont les mêmes
    if espece1 is espece2:
        return 0
    # Dans le cas où les deux espèces sont des espèces avérées
    if isinstance(espece2, EspeceAveree) and isinstance(espece1, EspeceAveree):
        return mutation_replacement_distance(espece1.get_genome(),
                                              espece2.get_genome())

    # Dans le cas où espece1 est hypothétique
    elif not isinstance(espece1, EspeceAveree):
        if not espece1.enfants:
            if not espece2.enfants:
                return 1.0
            return calculer_distance(espece1=espece2, espece2=espece1)

        somme_distances = 0
        for enfant in espece1.enfants:
            somme_distances += calculer_distance(enfant, espece2)
        return somme_distances / len(espece1.enfants)

    # Dans le cas où espece2 est hypothétique (et espece1 est avérée)
    elif not isinstance(espece2, EspeceAveree):
        return calculer_distance(espece1=espece2, espece2=espece1)

    return 0

```

## Question 12:

La construction d'un arbre phylogénétique automatique est un algorithme complexe, elle utilisera donc 2 méthodes.

La première méthode est **trouver\_paire\_minimale()**, cette méthode va retourner une paire d'espèce en calculant leur distance, on utilisera alors la méthode **calculer\_distance()** de la question 11 dans 2 for imbriqués qui est la manière la plus ergonomique sachant qu'il faut calculer la distance pour chaque espèce.

```
def trouver_paire_minimale(noeuds: list) -> tuple:
    """
    Trouve la paire d'espèces avec la distance minimale entre elles.
    """
    distance_min = None
    paire_min = None
    # for imbriqué afin de vérifier la distance sur tout les noeuds
    for noeud1 in noeuds:
        for noeud2 in noeuds:
            # tant que les 2 noeuds sont différents chacun de l'autre
            if noeud1 != noeud2:
                distance_init = calculer_distance(noeud1, noeud2)
                if distance_min is None or distance_min > distance_init:
                    distance_min = distance_init
                    paire_min = (noeud1, noeud2)
    return paire_min
```

La seconde méthode est **reconstruire\_arbre()** qui prend une liste d'espèces en paramètres, on initialise une variable qui va permettre de créer des parents entre deux espèces avérées en se servant de la méthode **trouver\_paire\_minimale()** pour récupérer la paire d'espèces avec la distance minimale.

Par la suite on crée une espèce hypothétique avec le nom des 2 espèces (nom et nom scientifique) pour en faire un parent, ce qui fait que l'on remplace la paire par le parent dans la liste de **noeuds\_actifs** et nous continuons jusqu'à n'avoir plus qu'une espèce qui sera la racine de l'arbre.

```
def reconstruire_arbre(liste_especes_de_depart: list) -> EspeceHypothetique:
    """
    Construit un arbre phylogénétique à partir d'une liste d'espèces
    (qui sont les feuilles de l'arbre).
    """

    noeuds_actifs = liste_especes_de_depart.copy()
    while len(noeuds_actifs) >= 2:
        paire_min = trouver_paire_minimale(noeuds_actifs)

        noeud1, noeud2 = paire_min
        noeuds_actifs.remove(noeud1)
        noeuds_actifs.remove(noeud2)
        #On crée leur parent
        nom_parent = f"({noeud1.get_nom()} et {noeud2.get_nom()})"
        nom_sci_parent = f"({noeud1.get_nom_sci()}, {noeud2.get_nom_sci()})"

        nouvel_ancetre = EspeceHypothetique(nom_parent, nom_sci_parent)
        nouvel_ancetre.add_enfants(noeud1)
        nouvel_ancetre.add_enfants(noeud2)

        noeuds_actifs.append(nouvel_ancetre)

    #On retourne la racine de l'arbre
    return noeuds_actifs[0]
```

Avec la méthode `_build_three_string()`, on peut retourner cette affichage sous forme d'exemple:

```
└─ L'espèce hypothétique (Insectivores et (Glires et (Xénarthres et (Afrothériens et Euarchonta)))) ((Insectivora, (Glires, (Xenarthra, (Afrotheria, Euarchonta))))
└─ L'espèce avérée Insectivores (Insectivora)
└─ L'espèce hypothétique (Glires et (Xénarthres et (Afrothériens et Euarchonta))) ((Glires, (Xenarthra, (Afrotheria, Euarchonta))))
└─ L'espèce avérée Glires (Glires)
└─ L'espèce hypothétique (Xénarthres et (Afrothériens et Euarchonta)) ((Xenarthra, (Afrotheria, Euarchonta)))
└─ L'espèce avérée Xénarthres (Xenarthra)
└─ L'espèce hypothétique (Afrothériens et Euarchonta) ((Afrotheria, Euarchonta))
└─ L'espèce avérée Afrothériens (Afrotheria)
└─ L'espèce avérée Euarchonta (Euarchonta)
```

# Tests :

Voici les résultats du coverage:

Coverage report: 83%

Files

Functions

Classes

coverage.py v7.11.3, created at 2025-11-15 21:04 +0100

File ▲	statements	missing	excluded	coverage
exploitation_echantillons\calculateur_distances.py	27	1	0	96%
exploitation_echantillons\constants.py	7	0	0	100%
exploitation_echantillons\echantillon.py	141	35	0	75%
exploitation_echantillons\espece_averree.py	13	1	0	92%
exploitation_echantillons\espece.py	32	0	0	100%
<b>Total</b>	<b>220</b>	<b>37</b>	<b>0</b>	<b>83%</b>

coverage.py v7.11.3, created at 2025-11-15 21:04 +0100