MASTER 1 MEEF : CAPES MATHS OPTION INFORMATIQUE

ECRIT BLANC - EPREUVE DE 3 HEURES

Les résultats des questions pourront être réutilisés ainsi que les différentes fonctions demandées non traitées au cours de l'épreuve. L'usage de la calculatrice est interdit. Le mémento Python est interdit.

Correction orthographique

Dans ce problème, on considère des mots sur un alphabet fini. La longueur d'un mot *u* est notée $|u|$. Les lettres d'un mot *u* de longueur *n* sont notées $u_0,\ldots,u_{n-1}$. Pour $0 \leq i \leq j \leq n$, on note $u_{i,j}$ le sous-mot de *u* de longueur $(j-i)$ constitué des lettres u_i, $u_{i+1},\ldots, u_{j-1}$. En particulier, $u = u_{0,n}$ et $u_{i,i}$ est le mot vide (aucune lettre) pour tout *i*.

Partie A : Distance entre des mots

La distance de Hamming entre deux mots *u* et *v* de même longueur, notée $\delta_H(u,v)$, est le nombre d'indices *i* tels que $u_i \neq v_i$.

A.1 Quelle est la distance de Hamming entre les mots "plage" et "glace" ?

La distance de Hamming entre "plage" et "glace" est 2 car il faut transformer le p en g et le g en c.

A.2 Si l'on ne considère que des mots écrits avec les 26 lettres minuscules non accentuées de l'alphabet latin, combien y a-t '-il de mots à distance au plus *d* d'un mot de longueur *n* donné ?

Supposons que $d \leq n$. Alors on peut effectivement modifier jusqu'à *d* lettres. Pour chacune, on a 26 possibilités (sachant que remettre la même lettre permet d'avoir une distance au plus *d*). Le nombre de mots à distance inférieure ou égale à d est donc $\binom{n}{d} 26^d$. Si $n < d$, on ne peut modifier que *n* lettres. Ainsi, le nombre total est 26^n si n<d et $\binom{n}{d} 26^d$ sinon.

A.3 Écrire la fonction **hamming** qui prend en entrée deux chaînes de caractère *u* et *v* et qui renvoie leur distance de Hamming $\delta_H(u,v)$. Cette fonction retournera **None** si les deux chaînes ne sont pas de même longueur.

```
def hamming(u, v):
    c = 0
    if len(u) != len(v):
        print("Chaînes de longueurs différentes.")
        return None
    for i in range(len(u)):
        if u[i] != v[i]:
            c += 1
    return c
```

La distance de Levenshtein entre deux mots *u* et *v* (de longueurs potentiellement distinctes), notée $\delta_L(u,v)$, est le nombre minimal de caractères qu'il faut supprimer, insérer ou modifier pour passer de *u* à *v*. Par exemple, la distance de Levenshtein entre les mots "pacte" et "plage" est 3 : partant de "pacte", il suffit

d'insérer un l en deuxième position ("placte"), de supprimer le t ("place"), et de remplacer le c par un g ("plage"). Il y a d'autres chemins possibles, mais on peut vérifier qu'il faut bien au moins 3 opérations pour passer d'un mot à l'autre.

A.4 Quelle est la distance de Levenshtein entre les mots "chapeau" et "crapaud" ?

La distance entre chapeau et crapaud est 3 : on remplace h par r, on supprime e et on ajoute d.
Bonus. On peut montrer que les mots sont à distance au moins 3 (pour l'instant, on n'a que la borne supérieure). Supposons qu'il existe une suite de (au plus) 2 opérations pour passer de chapeau à crapaud. Comme les deux mots ont la même longueur, s'il y a une suppression, il doit y avoir une insertion. S'il y a une suppression parmi les deux opérations, il faut nécessairement qu'il n'y ait comme autre opération qu'une insertion. Or sans remplacement, il faut supprimer le h et insérer le r. Mais alors il reste eau en fin de mot au lieu de aud. Si maintenant il n'y a ni suppression ni insertion, on obtient la distance de Hamming. Et ces deux mots sont à distance de Hamming 6 : les lettres à modifier dans chapeau sont h, e, a et u. Ainsi, les deux mots ne sont pas à distance inférieure ou égale à 2.

A.5 Montrer que pour tout *u* et *v* de même taille, $\delta_L(u,v) \leq \delta_H(u,v)$.

Comme indiqué dans la réponse précédente, on retrouve la distance de Hamming si on s'interdit les opérations d'insertion et de suppression. Cela démontre que $\delta_L(u,v) \leq \delta_H(u,v)$.

A.6 Montrer que pour tout *u* et *v*, $\delta_L(u,v) \leq \max(|u|,|v|)$.

Dans le pire des cas, on peut effectuer la suite d'opérations suivantes (en supposant que le mot le plus court soit u, sans perte de généralité) : on remplace toutes les lettres de u par les $|u|$ premières lettres de v, puis on insère en fin de mot les $(|v|-|u|)$ dernières lettres de v. On a une suite d'opérations de longueur $|v|$. Cela démontre le résultat, par symétrie, entre u et v.

A.7 Montrer que la distance de Levenshtein vérifie l'inégalité triangulaire : pour tout *u*, *v* et *w*,

$\delta_L(u,v) \leq \delta_L(u,w) + \delta_L(w,v)$.

Soit u, v et w. Pour passer de u à v, on peut commencer par transformer u en w, puis w en v. En prenant le nombre d'opérations optimal pour ces deux transformations, le nombre total d'opérations effectuées est $\delta_L(u,w) + \delta_L(w,v)$. Cela démontre l'inégalité.

Calcul naïf de la distance de Levenshtein :

A.8 Montrer que si $|u| = m$ et $|v| = n$,

$$\delta_L(u,v) = min \begin{cases} \delta_L(u_{1,m}, v) + 1 \\ \delta_L(u, v_{1,n}) + 1 \\ \delta_L(u_{1,m}, v_{1,n}) + 1_{u_0 \neq v_0} \end{cases}$$

où $1_{u_0 \neq v_0}$ vaut 1 si $u_0 \neq v_0$ et 0 sinon.

Considérons deux mots u et v, et une suite minimale de δ opérations pour passer de u à v. Ces opérations sont des remplacements, des insertions et des suppressions. Il est clair que ces opérations commutent : l'ordre dans lequel on les applique ne change rien. En effet, une même lettre de u ne peut pas être modifiée puis supprimée (cela contredirait la minimalité de la suite d'opérations), et de même une lettre ne peut pas être insérée pour être ensuite modifiée ou supprimée. On considère quatre cas en fonction de ce que devient la lettre u_0 dans v.

- Si u_0 reste en première position, alors les opérations ne touchent que le sous-mot $u_{1,m}$ pour donner $v_{1,n}$. On a donc $\delta = \delta_L(u_{1,m}, v_{1,n})$ dans ce cas.
- De même, si u_0 est transformée en une autre lettre v_0, les $(\delta-1)$ autres opérations transforment $u_{1,m}$ en $v_{1,n}$ et $\delta = 1 + \delta_L(u_{1,m}, v_{1,n})$.
- Si u_0 est supprimée, les $(\delta-1)$ autres opérations transforment $u_{1,m}$ en v, et $\delta = 1 + \delta_L(u_{1,m}, v)$.
- Enfin, si u_0 reste intacte mais que des lettres sont insérées avant, la première opération peut être l'insertion de v_0 devant u_0, et les autres opérations devront transformer u en $v_{1,n}$. On a alors $\delta = 1 + \delta_L(u, v_{1,n})$.

A.9 En déduire un algorithme récursif (naïf) de calcul de la distance de Levenshtein entre deux mots.

```
def levenshtein_naif(u,v):
    if len(u) == 0:
        return len(v)                    # insérer toutes les lettres
    if len(v) == 0:
        return len(u)                    # supprimer toutes les lettres
    d1 = levenshtein_naif(u[1:],v)
    d2 = levenshtein_naif(u,v[1:])
    d3 = levenshtein_naif(u[1:],v[1:])
    x = 1 if u[0] != v[0] else 0
    return min(1+d1, 1+d2, x+d3)
```

A.10 Analyser la complexité de votre algorithme.

Notons T(n) la complexité du calcul pour deux mots dont le plus court mesure une taille n. On effectue trois appels récursifs sur des tailles n−1. La complexité vérifie donc $T(n) \leq 3T(n-1)$. En posant $T(0)=c$ (le temps pour calculer la longueur des chaînes par exemple), on obtient $T(n) = 3^n \times c$ qui est en $O(3^n)$.

Bonus : On peut être un peu moins naïf dans l'algorithme en remarquant que si $u_0=v_0$, il suffit de faire un seul appel récursif `levenshtein(u[1:],v[1:])`. Cependant, si on a deux mots totalement différents (aucune lettre en commun par exemple), il y aura bien trois appels récursifs à chaque étape. Ainsi, on n'améliore pas la complexité dans le pire cas.

Afin d'améliorer la complexité de l'algorithme précédent, on recourt à la programmation dynamique. Soit *u* et *v* deux mots fixés. Pour $0 \leq i \leq m$ et $1 \leq j \leq n$, on note $\delta_{i,j} = \delta_L(u_{0,i}, v_{0,j})$.

A.11 Exprimer $\delta_L(u, v)$ en fonction d'un $\delta_{i,j}$.

$\delta_L(u, v) = \delta_{m,n}$

A.12 Pour $0 < i, j \leq n$, exprimer $\delta_{i,j}$ en fonction de différentes valeurs de $\delta_{i',j'}$, en s'inspirant de la question A.8.

D'après la question A.8, en raisonnant avec les mots à l'envers, on obtient

$$\delta_{i,j} = min \begin{cases} \delta_{i-1,j} + 1 \\ \delta_{i,j-1} + 1 \\ \delta_{i-1,j-1} + 1_{u_{i-1} = v_{j-1}} \end{cases}$$

A.13 Donner, en fonction de *i* et *j*, la valeur des cas de base $\delta_{i,0}$ et $\delta_{0,j}$.

$\delta_{i,0} = i$ et $\delta_{0,j} = j$ puisqu"il faut supprimer i lettres (resp. insérer j lettres) pour passer du mot vide à un mot de i lettres (resp. d'un mot de j lettres au mot vide).

A.14 Déduire des questions précédentes un algorithme **`levenshtein`** qui calcule la distance de Levenshtein entre deux mots, en remplissant itérativement un tableau qui contient la valeur $\delta_{i,j}$ en case *(i,j)*.

```
def levenshtein(u, v):
    m = len(u)
    n = len(v)
    delta = [[0] * (m+1) for _ in range(n+1)]
    delta[0] = list(range(m+1))
    for j in range(1, n+1):
        delta[j][0] = j
        for i in range(1, m+1):
            x = 0
            if u[i-1] != v[j-1]: x = 1
            delta[j][i] = min(delta[j][i-1] + 1, \
                              delta[j-1][i] + 1, \
                              delta[j-1][i-1] + x)
    return delta[-1][-1]
```

A.15 Quelles sont les complexités en temps et en mémoire de votre algorithme ?

Les complexités en mémoire et en temps sont identiques. On remplit un tableau de dimensions (m+1)×(n+1) où m et n sont les tailles respectives des entrées. Le remplissage de chaque case prend un nombre constant d'opérations arithmétiques (calcul du minimum de trois entiers). Les complexités en mémoire et en temps sont donc O(m × n).

Partie B : Correction de mots

Pour effectuer de la correction orthographique, on dispose d'un ensemble de mots valides (ex. issus du dictionnaire de la langue française). Cet ensemble est représenté sous la forme d'un arbre dont les arêtes sont étiquetées par des lettres de l'alphabet, et les nœuds peuvent être de deux natures : terminal ou non terminal. Un chemin de la racine vers un nœud définit naturellement un mot (la suite des lettres rencontrées), et l'ensemble des mots valides est donné par l'ensemble des chemins qui aboutissent sur un nœud terminal. Un nœud de l'arbre est identifié de manière unique par le chemin qui y conduit depuis la racine, ou de manière équivalente par le mot qu'il faut lire depuis la racine pour aboutir à ce nœud. Un exemple est donné par la figure 1.

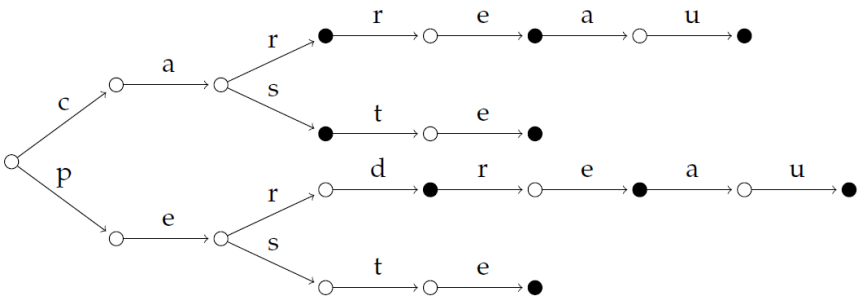

Figure 1: Exemple de représentation d'un ensemble de mots par un arbre. Les nœuds terminaux sont en noir, les non-terminaux en blanc. Par exemple, le mot "cas" appartient à l'ensemble car le nœud atteint en lisant "cas" est terminal, mais pas le mot "ca".

B.1 Donner la liste de tous les mots représentés par l'arbre de la figure 1.

Les mots représentés par l'arbre sont (de haut en bas, et du plus court au plus long) : car, carre, carreau, cas, caste, perd, perdre, perdreau, peste.

B.2 Montrer qu'il existe un unique arbre dont les feuilles sont terminales représentant un ensemble de mots donné.

On peut démontrer ce résultat par induction sur la longueur du plus long mot. Si le plus long mot est le mot vide, il existe un unique arbre le représentant : il est constitué d'un unique sommet qui est terminal (car toute feuille doit être terminale). Sinon, on peut partitionner l'ensemble des mots selon leurs premières lettres. Soit E l'ensemble de mots de départ, et pour toute lettre x, $E_x = \{w: xw \in E\}$, c'est-à-dire que E_x est l'ensemble des suffixes des mots de E commençant par la lettre x. Par hypothèse d'induction, il existe pour tout x un unique arbre A_x représentant E_x (ayant des feuilles terminales). On en déduit que l'unique représentation de E est obtenue en faisant l'union des A_x, en ajoutant une racine et un arc étiqueté x de la racine au sommet de A_x pour tout x.

B.3 Soit A un arbre représentant un ensemble de mots S. Supposons que la racine de l'arbre ait k fils $A_1,...,A_k$. Pour $1 \leq i \leq k$, on note x_i l'étiquette de l'arc allant de la racine de A à celle de A_i et S_i l'ensemble des mots représentés par A_i. Montrer que :

$$S = \bigcup_{i=1}^{k} \{u: u_0 = x_i, u_{1,|u|} \in S_i\}$$

Clairement, pour tout mot v $\in S_i$, le mot $x_i v$ appartient à S. Donc S contient l'union des $\{u: u_0 = x_i, u_{1,n} \in S_i\}$ où $n = |u|$. Soit maintenant u $\in$ S. Il existe un chemin dans A partant de la racine et aboutissant sur un nœud terminal dont la suite des étiquettes forme le mot u. Soit $x_i = u_0$. Alors après avoir lu la lettre x_i, on aboutit à la racine du sous-arbre A_i. Il existe donc un chemin dans A_i de la racine à un nœud terminal dont la suite d'étiquettes forme le mot $u_{1,n}$. On a donc démontré la deuxième inclusion, ce qui entraîne l'égalité souhaitée.

Les arbres décrits précédemment sont implémentés à l'aide de la classe **`Dico`** fournie ci-dessous.

```
class Dico:
    def __init__(self, terminal = False):
        self.terminal = terminal
        self.fils = {}

    def est_terminal(self):
        """Renvoie True si la racine de self est terminale."""
        return self.terminal

    def rend_terminal(self):
        """Modifie self pour rendre sa racine terminale."""
        self.terminal = True

    def lettres(self):
        """Renvoie la liste des lettres qui étiquettent un arc sortant de la
racine."""
        return list(self.fils.keys())

    def __getitem__(self, lettre):
        """Renvoie le sous-arbre pointé par l'arc d'étiquette lettre."""
        return self.fils[lettre]

    def greffe_dico(self, dico, lettre):
```

```
        """Ajoute le sous-arbre dico à self, avec un arc étiqueté lettre."""
        if lettre in self.lettres():
            print("La lettre existe déjà")
            return
        self.fils[lettre] = dico

    def ajout_lettre(self, lettre):
        """Ajoute un nouveau nœud non terminal comme fils de la racine, avec un
arc étiqueté lettre."""
        self.greffe_dico(Dico(), lettre)
```

La fonction **__getitem__** s'utilise de la manière suivante : si **d** est un objet de type **Dico** et **lettre** une des lettres étiquetant un arc sortant de la racine de **d**, alors **d[lettre]** (appel implicite à **__getitem__**) renvoie le sous-arbre de **d** qui se trouve à l'extrémité de l'arc étiqueté par **lettre**. Par exemple, si **d** est l'arbre de la figure 1, **d[c]** renvoie le sous-arbre constitué des 10 sommets « du haut » et **d[p]** renvoie le sous-arbre constitué des 11 sommets « du bas ».

Les méthodes suivantes doivent être écrites comme méthodes de la classe **Dico**. On rappelle que le premier paramètre des méthodes, habituellement noté **self**, est l'objet de type **Dico**.

B.4 Soit **d** l'objet de la classe **Dico** qui représente l'arbre de la figure 1. Que renvoient **d.est_terminal()** et **d.lettres()** ?

L'appel d.est_terminal() renvoie False car la racine de d n'est pas un nœud terminal. L'appel d.lettres() renvoie la liste [c, p] (éventuellement dans un autre ordre).

B.5 Ecrire une méthode **contient(self, mot)** qui prend en entrée un mot et renvoie vrai si le mot appartient au **Dico** et faux sinon.

```
def contient(self, mot):
    if len(mot) == 0:
        return self.est_terminal()
    c = mot[0]
    if c not in self.lettres():
        return False
    return self[c].contient(mot[1:])
```

B.6 Ecrire une méthode **liste_des_mots(self)** qui renvoie la liste des mots du **Dico**.

```
def liste_des_mots(self):
    L = []
    if self.est_terminal():
        L.append("")
    for lettre in self.lettres():
        liste = self[lettre].liste_des_mots()
        L.extend(lettre + mot for mot in liste)
    return L
```

B.7 Ecrire une méthode **ajout_mot(self, mot)** qui prend en entrée un mot et l'ajoute au **Dico**. Indication. La méthode ajoutera des nœuds à l'arbre si nécessaire, et modifiera un nœud existant sinon. Attention : cette méthode doit modifier **self** et ne rien renvoyer (de la même manière que **ajout_lettre**, qu'on utilisera).

```
def ajout_mot(self, mot):
    if len(mot) == 0:
```

```
        self.rend_terminal()
    else:
        if mot[0] not in self.lettres():
            self.ajout_lettre(mot[0])
        self[mot[0]].ajout_mot(mot[1:])

def ajout_mots(self, liste):
    for mot in liste:
        self.ajout_mot(mot)
```

B.8 Ecrire une méthode récursive **correction_hamming(self, mot, k)** qui prend en entrée un mot et un entier k, et renvoie la liste des mots du **Dico** à distance au plus k de mot.
Indication. Il faut parcourir les lettres d'un nœud et ajouter à la liste résultat les mots des sous-arbres préfixés par chacune de ces lettres (la distance étant inchangée si la lettre est le premier caractère du mot et décrémentée sinon). Si on arrive sur un mot vide, alors la liste résultat est vide si le nœud est non terminal et contient un mot vide sinon.

```
def correction_hamming(self, mot, k):
    if len(mot) == 0:
        return [""] if self.est_terminal() else []
    L = []
    for c in self.lettres():
        if c == mot[0]:
            corriges = self[c].correction_hamming(mot[1:], k)
            L.extend(c + corr for corr in corriges)
        elif k > 0:
            corriges = self[c].correction_hamming(mot[1:], k-1)
            L.extend(c + corr for corr in corriges)
    return L
```

B.9 (Question bonus) Ecrire une méthode **correction_levenshtein(self, mot)** qui prend en entrée un mot et renvoie le **Dico** contenant exactement les mots à distance de Levenshtein au plus 1 de mot.
Indication. Une distance de Levenshtein de 1 signifie qu'une lettre est modifiée, ajoutée ou supprimée. En itérant sur les mots présents, on peut récupérer ces mots (ajout en fin ou à la place de chaque lettre si possible, suppression d'une lettre lorsqu'un sous-mot existe, modification d'une lettre lorsqu'un sous-mot existe).

```
def correction_levenshtein(self, mot):
    dico = Dico()
    if len(mot) == 0:
        if self.est_terminal():
            dico.ajout_mot("") # mot vide correct
        for c in self.lettres():
            if self[c].est_terminal():
                dico.ajout_mot(c) # ajout d'une lettre
        return dico

    if self.contient(mot[1:]): # suppression d'une lettre
        dico.ajout_mot(mot[1:])

    for c in self.lettres():
        if c == mot[0]: # 1ere lettre correcte
            corriges = self[c].correction_levenshtein(mot[1:])
            dico.greffe_dico(corriges, c)
        else:
            if self[c].contient(mot[1:]):
```

```
                dico.ajout_mot(c + mot[1:]) # modification d'une lettre
            if self[c].contient(mot):
                dico.ajout_mot(c + mot)     # ajout d'une lettre
    return dico
```

Partie C : Compression du `Dico`

On souhaite compresser l'arbre en le représentant maintenant par un graphe orienté acyclique. Pour cela, on note que certains nœuds sont « équivalents » s'ils sont de même nature (terminal ou non terminal), et qu'ils ont les mêmes sous-arbres (avec les mêmes étiquettes). Par exemple, dans l'arbre de la figure 1, les nœuds atteints en lisant "car" et "perd" sont équivalents. En fusionnant les nœuds équivalents, on obtient le graphe orienté acyclique de la figure 2.

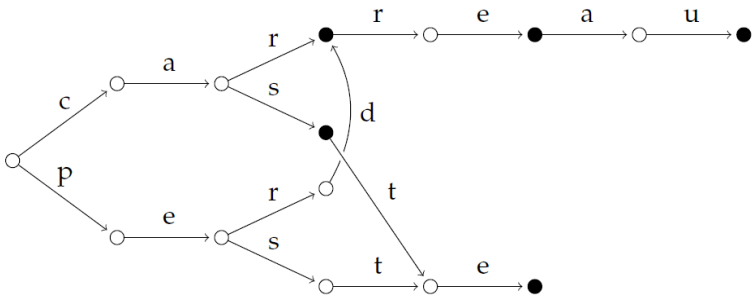

Figure 2: Exemple de représentation d'un ensemble de mots par un graphe orienté acyclique.

C.1 Quelle est la liste des mots représentés par le graphe orienté acyclique de la figure 2 ?

Le graphe de la figure 2 représente exactement le même ensemble de mots que l'arbre de la figure 1.

C.2 Pourquoi ne peut-on pas fusionner les nœuds atteints en lisant "ca" et "pe" ?

Les nœuds atteints en lisant "ca" et "pe" sont bien de même nature, mais leurs sous-arbres sont différents. Par exemple en lisant un r, on atteint un nœud terminal dans un cas et non-terminal dans l'autre.

C.3 Montrer que deux nœuds ne peuvent pas être équivalents si l'un est l'ancêtre de l'autre.

Par définition, un nœud et l'un de ces ancêtres ont des sous-arbres différents puisque le nœud appartient à l'un des sous-arbres de son ancêtre, mais n'appartient pas à l'un de ses propres sous-arbres.

C.4 Montrer que fusionner deux nœuds équivalents ne modifie pas l'ensemble des mots représentés.

Deux nœuds sont équivalents si pour tout mot w, quand on suit les chemins étiquetés par w depuis l'un ou l'autre des nœuds, on arrive à un nœud de même nature (ou éventuellement on ne peut pas lire w jusqu'au bout dans les deux cas). Ainsi, en fusionnant les nœuds, on ne change pas le fait que w appartienne à l'ensemble ou non. La liste des mots reste donc inchangée.

Pour compresser le **`Dico`**, on utilise la méthode suivante :

1. on partitionne les nœuds en classes d'équivalences
2. on fusionne tous les nœuds d'une même classe d'équivalence

On cherche à implémenter cette méthode dans notre classe. On va identifier chaque nœud de l'arbre avec le mot qui permet de l'atteindre.

C.5 Écrire une méthode **tous_les_mots(self)** qui renvoie la liste de tous les mots présents dans l'arbre, qu'ils appartiennent ou non à l'ensemble de mots valides.
Indication. En appliquant la méthode à l'arbre de la figure 1, on doit obtenir une liste de 22 mots, commençant par **["", "c", "p", "ca", "pe", ...]**.

```
def tous_les_mots(self):
    L = [""]
    for l in self.lettres():
        L.extend(l + m for m in self[l].tous_les_mots())
    return L
```

C.6 Écrire une méthode **parcours(self, mot)** qui renvoie le **Dico** (i.e. le nœud) atteint lorsqu'on lit le **mot**.

```
def parcours(self, mot):
    if len(mot) == 0:
        return self
    c = mot[0]
    if c not in self.lettres():
        print("Le mot n'apparaît pas dans l'arbre")
    return self[c].parcours(mot[1:])
```

C.7 Écrire une méthode **est_equivalent(self, dico)** qui teste si le **Dico self** est équivalent au **Dico dico** (mêmes nœuds de même nature).

```
def est_equivalent(self, dico):
    if self.est_terminal() != dico.est_terminal():
        return False
    l_self = self.lettres()
    l_dico = dico.lettres()
    if len(l_self) != len(l_dico):
        return False
    for lettre in l_self:
        if lettre not in l_dico:
            return False
        if not self[lettre].est_equivalent(dico[lettre]):
            return False
    return True
```

C.8 (Question bonus) Écrire une méthode **compresse(self)** qui compresse le **Dico** grâce à l'algorithme décrit précédemment.
Indication. Numéroter les nœuds en utilisant l'indice du mot correspondant dans la liste renvoyée par **tous_les_mots**. Tester l'équivalence des nœuds (sous-arbres) deux à deux, et utiliser comme représentant unique d'une classe d'équivalence, le nœud d'indice le plus faible. Une fois les classes d'équivalence calculées, remplacer chaque nœud par le représentant de sa classe d'équivalence.

```
def compresse(self):
    mots = self.tous_les_mots()                  # tous les mots existants
    parties = list(range(len(mots)))             # liste des indices
    for i in range(len(mots)):                   # pour tous les mots
        dico1 = self.parcours(mots[i])    # placement sur le dernier noeud de i
        for j in range(i+1, len(mots)):          # pour toutes les paires
            dico2 = self.parcours(mots[j])     # placement sur mot j
            if dico1.est_equivalent(dico2):      # si meme sous-arbre
                parties[j] = parties[i]          # maj du représentant
    for i in range(len(mots)):                   # pour tout les mots
```

```
        if parties[i] < i:                              # si compression possible
            dico1 = self.parcours(mots[i][:-1])   # placement sur av-der. mot i
            dico2 = self.parcours(mots[parties[i]])  # placement sur repres. i
            dico1.fils[mots[i][-1]] = dico2          # maj noeud
```