Partie I

1. a) Seuls les caractères d'indices 0 et 1 sont différents donc la distance de Hamming de 'pcsi' à 'mpsi' est 2
 b) Une boucle suffit, ne pas oublier l'hypothèse $\text{len}(\text{ch1})=\text{len}(\text{ch2})$:

```
def Hamming(ch1, ch2) :
    d = 0
    for i in range(len(ch1)) :
        if ch1[i] != ch2[i] :
            d += 1
    return d
```

- c) Une seule boucle donc $C(n) = O(n)$; complexité linéaire
 2. a) On peut passer de 'mpii' à 'psi' avec 2 transformations :

$$'mpii' \xrightarrow{\text{supp}} 'pii' \xrightarrow{\text{modif}} 'psi'$$

donc la distance est au plus 2. Il n'est pas possible de le faire avec une seule transformation car 'mpii' est de longueur 4 alors que 'psi' est de longueur 3 donc il faut obligatoirement une suppression et une suppression seule ne peut pas suffire. On a donc $d_{lev}('mpii', 'psi') = 2$

- b) On a
- $d_{0,j} = j$ et $d_{i,0} = i$ il faut faire j (ou i) ajouts
 - si $\text{ch1}[i-1]=\text{ch2}[j-1]$, $d_{i,j} = d_{i-1,j-1}$: les modifications sont à faire sur $\text{ch1}[:i-1]$ et $\text{ch2}[:j-1]$ sans modifier le dernier caractère de $\text{ch1}[:i]$ et $\text{ch2}[:j]$
 - Sinon $d_{i,j} = 1 + \min\{d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}\}$: $1 + d_{i-1,j}$ correspond à supprimer le dernier caractère de $\text{ch1}[:i]$, $1 + d_{i,j-1}$ à supprimer le dernier caractère de $\text{ch2}[:j]$ (ou à l'ajouter à la fin de $\text{ch1}[:i]$) et $1 + d_{i-1,j-1}$ à remplacer le dernier caractère de $\text{ch1}[:i]$ par le dernier de $\text{ch2}[:j]$ (qui est $\text{ch2}[j-1]$).
- c) Dans le cas de deux chaînes de longueurs n (égales pour simplifier) qui n'ont aucun caractère en commun, chaque appel à la fonction va récursivement appeler 3 fois la même fonction avec des chaînes de longueurs $\geq n-1$ donc la complexité sera $\geq 3^n$ (exponentielle). L'inefficacité de cette fonction provient des problèmes de recouvrement : on va exécuter de nombreuses fois la fonction avec les mêmes sous chaînes.
- d) On initialise D avec la bonne première ligne (et la bonne taille) puis on modifie les lignes d'indices ≥ 1 les unes après les autres :

```
def tableauLEV(ch1, ch2) :
    n,m = len(ch1), len(ch2)
    D = [[j for j in range(m+1)] for i in range(n+1)]
    for i in range(1, n+1) :
        D[i][0] = i
        for j in range(1, m+1) :
            if ch1[i-1] == ch2[j-1] :
                D[i][j] = D[i-1][j-1]
            else :
                D[i][j] = 1+min([D[i-1][j], D[i][j-1], D[i-1][j-1]])
    return D
```

- e) Il suffit de récupérer la valeur de la case en bas à droite du tableau; pour une liste, l'élément d'indice -1 est le dernier élément de cette liste :

```
def distLEV(ch1, ch2) :
    D = tableauLEV(ch1, ch2)
    return D[-1][-1]
```

- f) la fonction `tableauLEV` a une complexité en $O(nm)$: l'initialisation du tableau D est déjà en $O(nm)$ puis la double boucle imbriquée aussi; les deux opérations étant consécutives, la complexité globale est $O(nm)$. La sélection du dernier élément du tableau est négligeable devant nm (elle est en $O(m)$ car $D[-1]$ crée une liste de longueur $m+1$) donc $C(n, m) = O(nm)$
 g) Il faut trouver un chemin permettant de remonter de la case finale jusqu'à un des bords du tableau (en haut ou à gauche), il reste ensuite seulement à faire des ajouts ou des suppression selon le bord atteint. Le problème provient de certaines ambiguïtés :

- pour $i = 8$ et $j = 7$, on a $d_{i,j} = 1 + d_{i-1,j} = 1 + d_{i-1,j-1}$: la listes des transformations à appliquer n'est pas unique, on peut commencer soit par supprimer le dernier caractère de `ch1`, soit le remplacer par le dernier caractère de `ch2` mais ça ne posera pas de problème par la suite.
 - pour $i = 3$ et $j = 3$, on a $d_{i,j} = d_{i-1,j-1} = 1 + d_{i-1,j}$: il semble que l'on puisse indifféremment supprimer le dernier caractère de `ch1[:i]` ou le laisser tel quel est continuer. Mais cette dois, comme `ch1[:i]='dis'` et `ch2[:j]='édi'`, l'égalité $d_{i,j} = d_{i-1,j-1}$ ne provient pas de l'égalité `ch1[i-1]=ch2[j-1]`.
- Un chemin exploitable est donc marqué en gras

0	1	2	3	4	5	6	7
1	1	1	2	3	4	5	6
2	2	2	1	2	3	4	5
3	3	3	2	2	3	4	5
4	4	4	3	2	3	4	5
5	5	5	4	3	3	4	5
6	6	6	5	4	4	4	4
7	7	7	6	5	5	5	5
8	8	8	7	6	6	6	6

On effectue donc successivement deux suppressions, une insertion, une modification, une suppression (ce qui amène au bord supérieur) et on rajoute la dernière lettre manquante (la première de `ch2`) : au total on a bien 6 opérations.

- h) L'analyse précédente étant bien faite, il suffit de faire attention à tester plutôt si `ch1[i-1]==ch1[j-1]` pour distinguer les cas, plutôt que juste utiliser les coefficients du tableau :

```
def transfLEV(ch1, ch2) :
    d = tableauLEV(ch1, ch2)
    L = [ch1]
    i, j = len(ch1), len(ch2)
    suffixe = ''
    while i>0 and j>0 :
        if ch1[i-1] == ch2[j-1] : # caractère commun : rien à faire
            i, j = i-1, j-1
        elif d[i][j] == 1+d[i-1][j] : # une suppression
            ch1 = ch1[:i-1]+ch1[i:]
            L.append(ch1)
            i = i-1
        elif d[i][j] == 1+d[i][j-1] : # une insertion
            ch1 = ch1[:i]+ch2[j-1]+ch1[i:]
            L.append(ch1)
            j = j-1
        else :
            ch1 = ch1[:i-1]+ch2[j-1]+ch1[i:] # une modification
            L.append(ch1)
            i, j = i-1, j-1
    # on est au bord
    while i>0 : # rien ne se passe si i=0, sinon supprime les lettres en
        trop
        ch1 = ch1[:i-1]+ch1[i:]
        L.append(ch1)
        i = i-1
    while j>0 : # et ici on rajoute celles qui manquent
        ch1 = ch2[j-1]+ch1
        L.append(ch1)
        j = j-1
    return L
```

- i) On utilise un dictionnaire dont les clefs sont les couples (`ch1[:i], ch2[:j]`) (on pourrait utiliser seulement les couples (i, j)) :

```
def LEV(ch1, ch2) :
    d = {}
    def Lev(ch1, ch2) :
        if (ch1, ch2) not in d :
            n, m = len(ch1), len(ch2)
            if n*m == 0 :
```

```

        r = n+m
    else :
        d1 = Lev(ch1[:n-1],ch2[:m-1])
        if ch1[n-1] == ch2[m-1] :
            r = d1
        else :
            d2 = Lev(ch1, ch2[:m-1])
            d3 = Lev(ch1[:n-1],ch2)
            r = 1+min(d1, d2, d3)
    d[(ch1, ch2)] = r
    return d[(ch1, ch2)]
return Lev(ch1, ch2)

```

Partie II

1. a) 'cap', 'copie', 'copier', 'copies', 'cor', 'corde', 'corne', 'correct', 'correcte'

b) La fonction compte le nombre d'étiquette '\$' donc le nombre de mots valides

```

c) def contient(mot, dico) :
    if len(mot) == 0:
        return '$' in dico
    else :
        lettre = mot[0]
        if lettre in dico : # la première lettre du mot est une étiquette
            mot = mot[1:]
            dico = dico[lettre]
            return contient(mot, dico) # on vérifie le reste du mot dans le
                sous arbre correspondant
        else :
            return False # la première lettre n'est déjà pas une étiquette

```

d) Plus facile à écrire récursivement (comme toutes les fonctions sur de tels objets) :

```

def branche(mot) :
    if len(mot) == 0 :
        return {'$' : {}}
    else :
        lettre = mot[0] # première étiquette
        d = branche(mot[1:]) # création récursive du sous arbre
            correspondant
    return {lettre : d}

```

e) On « descend » dans l'arbre jusqu'à trouver la position où ajouter la fin du mot, en « greffant » la branche restante :

```

def ajoutMot(mot, dico) :
    if len(mot) == 0 : # le mot y est déjà mais peut être pas marqué valide
        dico['$'] = {}
    else :
        lettre = mot[0]
        if lettre not in dico : # on a trouvé la première lettre non pré
            sente
            dico[lettre] = branche(mot[1:])
        else :
            ajoutMot(mot[1:], dico[lettre]) # sinon on descend dans le sous
                arbre en enlevant la première lettre de mot

```

2. a) Un mot à distance ≤ 1 est obtenu avec une suppression, un ajout ou une modification au plus.

'pp' $\xrightarrow{\text{modif}}$ 'mp' (ou 'pc')

'oc' $\xrightarrow{\text{modif}}$ 'pc'

'ps' $\xrightarrow{\text{ajout}}$ 'psi' (ou $\xrightarrow{\text{modif}}$ 'pc')

'pps' impossible

'psii' $\xrightarrow{\text{supp}}$ 'psi'

- b) On regarde si la première lettre du mot fait parti du dictionnaire et dans ce cas, on cherche (récursivement) un suffixe du dictionnaire à distance ≤ 1 s'il existe : si la fonction renvoie `None` c'est qu'il n'y en a pas. Si la première lettre n'est pas valide, il faut la supprimer et le reste du mot doit être dans le dictionnaire, ou bien il faut la modifier par une lettre valide qui rajouté à la suite du mot doit donner un mot valide du dictionnaire.

```
def corLev(mot, dico) :
    if len(mot) == 0 :
        return ''
    else :
        lettre = mot[0]
        if lettre in dico : # on garde la première lettre
            mot = mot[1:]
            d = dico[lettre]
            suffixe = corLev(mot,d)
            if suffixe != None : # on a trouvé un moyen de la compléter
                return lettre+suffixe
        else :
            mot = mot[1:]
            if contient(mot,dico) : # il suffit de supprimer la première
                lettre
                return mot
            else :
                for k in dico : # on cherche à remplacer la première lettre
                    if contient(mot,dico[k]) : # la fin du mot doit être
                        dans le sous arbre correspondant
                        return k+mot
```