Gestion de versions de grands textes

Partie I : Différentiels par positions fixes

Question 1. La fonction `textes_égaux` a une complexité linéaire c'est à dire $\mathcal{O}$(`len(texte1)`).

```
def textes_égaux(texte1, texte2) :
    n = len(texte1) # longueur commune
    i = 0
    while i < n and texte1[i] == texte2[i] :
        i += 1
    return i == n
```

Question 2. La fonction `distance(texte1, texte2)` a une complexité linéaire c'est à dire $\mathcal{O}$(`len(texte1)`).

```
def distance(texte1, texte2) :
    n = len(texte1) # longueur commune
    dist = 0
    for i in range(n) :
        if texte1[i] != texte2[i] :
            dist += 1
    return dist
```

Question 3. Dans la fonction `aucun_caractère_commun(texte1, texte2)`, on se sert du dictionnaire `dic` comme d'un ensemble. En effet l'ensemble de ses clés est l'ensemble des caractères de `texte1`. La construction du dictionnaire a une complexité $\mathcal{O}$(`len(texte1)`).

Ainsi avec la deuxième boucle cette fonction a une complexité $\mathcal{O}$(`len(texte1)` + `len(texte2)`).

```
def aucun_caractère_commun(texte1, texte2) :
    n1 = len(texte1)
    n2 = len(texte2)
    dic = { }
    for i in range(n1) :
        dic[texte1[i]] = True
    j = 0
    while j < n2 and not(texte2[j] in dic) :
        j += 1
    return j == n2
```

Question 4. Dans la fonction `différentiel(texte1, texte2)`, lors du parcours simultané des textes le `egalcar` signale si le caractère précédent correspondait à une tranche ou pas.

Si `egalcar` vaut `True` et que les caractères diffèrent, on « ouvre » une nouvelle tranche.

Si `egalcar` vaut `False` et que les caractères sont égaux, on « ferme » la tranche courante.

À la fin du parcours, on ferme la dernière tranche si `egalcar` vaut `False`.

La complexité de `différentiel(texte1, texte2)` est bien $\mathcal{O}$(`len(texte1)`) car il y a une seule boucle et que chaque tour de boucle s'effectue en temps constant (au plus 6 opérations élémentaires).

```
def différentiel(texte1, texte2) :
    n = len(texte1)
    egalcar = True
    arg_debut = 0
    arg_avant = []
    arg_après = []
    out = []
    for i in range(n) :
        if egalcar :
            if texte1[i] != texte2[i] :
                arg_debut = i
                arg_avant.append(texte1[i])
                arg_après.append(texte2[i])
                egalcar = False
        else :
            if texte1[i] != texte2[i] :
                arg_avant.append(texte1[i])
                arg_après.append(texte2[i])
            else :
                out.append(tranche(arg_debut, arg_avant, arg_après))
                arg_avant = []
                arg_après = []
                egalcar = True
    if not egalcar :
        out.append(tranche(arg_debut, arg_avant, arg_après))
    return out
```

Remarque : je me sers uniquement des fonctions de l'interface associée à la structure de données de *tranches*. Ce sont les fonctions fournies par l'énoncé. L'aspect *immuable* des tranches est respectées.

Question 5. Lors de l'exécution de `applique(texte1, diff)` on commence par faire une copie `texte1` de complexité $\mathcal{O}$`(len(texte1))`. L'exécution de `apptr(texte, tr)` qui applique une tranche d'un différentiel au texte d'origine est de complexité $\mathcal{O}$`(len(tr))`. La somme de longueurs des tranches étant majorée par `(len(texte1))`, la complexité de `applique(texte1, diff)` est de complexité $\mathcal{O}$`(len(texte1))`.

```
def apptr(texte, tr) :# applique tranche à texte
    for i in range(début(tr),fin(tr)) :
        texte[i] = après(tr)[i-début(tr)]

def copie(texte) :
    textcop =[]
    n = len(texte)
    for i in range(n) :
        textcop.append(texte[i])
    return textcop

def applique(texte1, diff) :
    k = len(diff)
    texte2 = copie(texte1)
    for j in range(k) :
        apptr(texte2, diff[j])
    return texte2
```

Question 6. Il suffit d'inverser sur chaque tranche les valeurs des clés `'avant'` et `'après'`.

La complexité de `inverse(diff)` est $\mathcal{O}$(`len(diff)`).

```
def inverse(diff) :
    k = len(diff)
    diffs = []
    for i in range(k) :
        tr = diff[i]
        trs = tranche(début(tr), après(tr), avant(tr))
        diffs.append(trs)
    return diffs
```

Question 7. Les fonctions `modifie(texte_versionné, texte)` et `annule(texte_versionné)` ont respectivement comme complexité $\mathcal{O}$(`len(texte)`) où pour la deuxième fonction `texte` est le retour de la fonction.

```
def modifie(texte_versionné, texte) :
    diff = différentiel(courant(texte_versionné), texte)
    remplace_courant(texte_versionné, texte)
    historique(texte_versionné).append(diff)

def annule(texte_versionné) :
    diff = historique(texte_versionné).pop()
    texte = applique(courant(texte_versionné), inverse(diff))
# texte est le nouveau texte courant
    remplace_courant(texte_versionné, texte)
    return texte
```

Partie II : Différentiels sur des positions variables

Question 8. La fonction `poids(diff)` a une complexité de $\mathcal{O}$(`len(diff)`).

```
def poids(diff) :
    pds = 0
    nd = len(diff)
    for i in range(nd) :
        tr = diff[i]
        pds += len(avant(tr))
        pds += len(après(tr))
    return pds
```

Question 9. On considère $0 \leqslant i <$ `ln(texte`$_1$`)` et $0 \leqslant j <$ `ln(texte`$_2$`)`. Il y a deux cas disjoints :

Premier cas : Si `texte`$_1$`[i]` $=$ `texte`$_2$`[j]`, alors on a $\mathrm{M}[i+1][j+1] = \mathrm{M}[i][j]$

Deuxième cas : Sinon, on a : $\mathrm{M}[i+1][j+1] = 1 + \min\left(\mathrm{M}[i][j+1], \mathrm{M}[i+1][j]\right)$

Pour le premier cas, la distance d'édition ne changera pas en ajoutant une même lettre située en dernière position. Dans le deuxième cas, « $+1$ » correspond au rajout d'une dernière lettre. En effet, on rajoute un caractère à un des textes distinct du dernier de l'autre de sorte que la distance d'édition entre `texte`$_1$`[ :i+1]` et `texte`$_2$`[ :j+1]` est le minimum de $\begin{cases} 1 + \text{celle entre } \texttt{texte}_1\texttt{[:i]} \text{ et } \texttt{texte}_2\texttt{[:j+1]} \\ 1 + \text{celle entre } \texttt{texte}_1\texttt{[:i+1]} \text{ et } \texttt{texte}_2\texttt{[:j]} \end{cases}$.

Question 10. On commence par écrire une fonction qui calcule le minimum pour deux arguments entiers.

On choisit une programmation dynamique de haut en bas.

Toutefois il s'agit de ne pas oublier les cas de base (coefficients non donnés par la relation de récurrence). Ces coefficients de la forme M[i][0] ou M[0][j] valent $i+j$. Le coefficient particulier M[0][0] $= 0$ est calculé deux fois par la fonction interne pour `calculcoef(0,0)` et `calculcoef(1,1)` et c'est le seul.

```
def mini2(a,b) :
    if a <b : return a
    else : return b


def levenshtein_desc(texte1, texte2) :
    n = len(texte1) + 1
    m = len(texte2) + 1
    M = [[0 for j in range(m)] for i in range(n) ]
    def calculcoef(i,j) :
        if i*j == 0 :
            M[i][j] = i+j
        else : #on force le calcul de tous les coefficients
            if M[i-1][j-1] == 0 : M[i-1][j-1] = calculcoef(i-1,j-1)
            if M[i][j-1] == 0 : M[i][j-1] = calculcoef(i,j-1)
            if M[i-1][j] == 0 : M[i-1][j] = calculcoef(i-1,j)
            if texte1[i-1] == texte2[j-1] : M[i][j] = M[i-1][j-1]
            else : M[i][j] = 1+ mini2(M[i-1][j], M[i][j-1])
        return M[i][j]
    calculcoef(n-1,m-1)
    return M
```

La fonction interne `calculcoef(i,j)` est appelée 1+`(len(texte1))`×`(len(texte2))` fois.

Le nombre de tests et opérations unitaires hors appels récursifs de la fonction interne est en $\mathcal{O}(1)$ (majoré par une constante). Le nombre d'opérations pour l'initialisation pour `M` est aussi en $\mathcal{O}$`(len(texte1))`×`(len(texte1))`.

Ainsi `levenshtein(texte1, texte2)` est de complexité polynomiale $\mathcal{O}$`(len(texte1))`×`(len(texte1))`.

Néanmoins une programmation dynamique ascendante ne paraît pas très naturelle par rapport à une descendante ci dessous. La complexité est clairement là même.

```
def levenshtein_asc(texte1, texte2) :
    n = len(texte1) + 1
    m = len(texte2) + 1
    M = [[0 for j in range(m)] for i in range(n) ]
    for i in range(0,n) :
        for j in range(0,m) :
            if i == 0 : M[i][j] = j
            elif j == 0 : M[i][j] = i
            elif texte1[i-1] == texte2[j-1] :
                M[i][j] = M[i-1][j-1]
            else : M[i][j] = 1+ mini2(M[i-1][j], M[i][j-1])
    return M
```

Question 11. La première fonction `inverse(liste)` renvoie le mirroir de la liste argument.

Cette fonction a clairement une complexité en $\mathcal{O}$(`len(liste)`).

```
def inverse(liste):
    n = len(liste)
    return [liste[n-i] for i in range(1, n+1)]
```

La seconde fonction consiste à parcourir la matrice M à l'envers en partant de la position (n, m) ($n =$`len(texte1)`, $m =$`len(texte2)`), on s'arrête en $(0, 0)$. La variable `sur_tranche` est un booléen signalant qu'on remplit une tranche ; on l'utilise de façon analogue à la question 4.

Si $i > 0$ et $j > 0$ et $\mathrm{M}[i][j] \notin \{\mathrm{M}[i-1][j]+1, \mathrm{M}[i][j-1]+1\}$, alors $\mathrm{M}[i][j] = \mathrm{M}[i-1][j-1]$ et on se trouve pas sur une tranche. Lors du déplacement correspondant, le coefficient ne varie pas.

Sinon, on se retrouve sur une tranche et on remplit, à l'envers une des deux listes `arg_avant` ou bien `arg_après`. Dans ce cas, le coefficient varie de 1.

Ainsi à chaque étape, si le poids du différentiel en devenir et le coefficient correspondant de la matrice varie simultanément et la variation est identique.

Ainsi le différentiel calculé satisfait les propriétés attendues car son poids vaut la distance de Levenshtein entre `texte1` et `texte2`.

```
def différentiel(texte1, texte2, M):
    i, j = len(texte1), len(texte2)
    sur_tranche = False #indique que l'on remplit une tranche
    out = []
    while i>0 or j >0:
        if i>0 and j>0 and texte1[i-1] == texte2[j-1]:
            if sur_tranche:
                tr = tranche(i, inverse(arg_avant), j, inverse(arg_après))
                out.append(tr)
                sur_tranche = False
            i, j = i-1, j-1
        else:
            if not sur_tranche:
                sur_tranche = True
                arg_avant, arg_après = [], []
            if i>0 and j >0:
                if  M[i][j-1] <= M[i-1][j]:
                    j -= 1
                    arg_après.append(texte2[j])
                else:
                    i -= 1
                    arg_avant.append(texte1[i])
            elif i==0:
                j -= 1
                arg_après.append(texte2[j])
            else:
                i -= 1
                arg_avant.append(texte1[i])
    if sur_tranche:
        tr = tranche(i, inverse(arg_avant), j, inverse(arg_après))
        out.append(tr)
    return inverse(out)
```

Le nombre d'opérations par tour de boucle est majoré par une constante auquel il faut ajouter les tailles des listes `arg_avant` et `arg_après`. La somme des tailles de ces listes est majorée par $n+m$ et il en est de même pour la liste `out`. Les différentes utilisations de la fonction `inverse` ont au total une complexité $\leqslant n+m$. Ainsi la fonction `différentiel(texte1, texte2, M)` a une complexité $\mathcal{O}($`len(texte1)` $+$ `len(texte2)`$)$.

Question 12. On utilise le fait que les tranches sont rangées dans l'ordre croissant des positions.

```
def conflit(diff1, diff2):
    n1, n2 = len(diff1), len(diff2)
    i1, i2 = 0, 0
    while i1 < n1 and i2 < n2:
        tr1, tr2 = diff1[i1], diff2[i2]
        if début_avant(tr1) < début_avant(tr2):
            if fin_avant(tr1) >= début_avant(tr2):
                return True
            else:
                i1 += 1
        elif début_avant(tr2) < début_avant(tr1):
            if fin_avant(tr2) >= début_avant(tr1):
                return True
            else:
                i2 += 1
        else:
            return True
    return False
```

Le nombre de tours de boucles est majoré par `(len(diff1)+len(diff1))`. Chaque boucle s'effectue en temps constant. La fonction `conflit(diff1, diff2)` a donc bien une complexité en $\mathcal{O}$`(len(diff1)+len(diff1))`.

On remarque que la condition de non-conflit de l'énoncé est un peut strict mais elle évite les ambiguïtés si la liste `avant(tr)` est vide.

Question 13. La variable `trans` indique de combien translater les tranches de `diff2`.

```
def fusionne(diff1, diff2):
    n1, n2 = len(diff1), len(diff2)
    i1, i2 = 0, 0
    out = []
    trans = 0
    while i2 < n2:
        tr2 = diff2[i2]
        while i1 < n1 and fin_avant(diff1[i1]) <  début_avant(tr2):
            trans += len(après(diff1[i1])) - len(avant(diff1[i1]))
            i1 += 1
        trnew = tranche(début_avant(tr2)+trans, avant(tr2), début_après(tr2)+trans, après(tr2))
        out.append(trnew)
        i2 += 1
    return out
```

Cette fonction a bien une complexité $\mathcal{O}$`(len(diff1)+len(diff1))`.

Partie III : Calcul de différentiels par calcul de plus courts chemins

Question 14. Expliquons ce résultat par récurrence double.

Initialisation sur i : On suppose $i = 0$.

Initialisation sur j : Pour $j = 0$, le sommet $(0, 0)$ se situe à une distance de $\mathrm{M}[0][0] = 0$ de $(0, 0)$.

Hérédité sur j : Soit $j \in [\![0, m-1]\!]$ tel que $(0, j)$ se situe à une distance de $\mathrm{M}[0][j] = j$ de $(0, 0)$.
Le sommet $(0, j+1)$ admet $(0, j)$ comme unique prédécesseur
donc sa distance à $(0, 0)$ vaut $1 + \mathrm{M}[0][j] = j + 1 = \mathrm{M}[0][j+1]$.

Conclusion sur j : Ainsi pour tout $j \in [\![0, m]\!]$, le sommet $(0, j)$ se situe à une distance $\mathrm{M}[0][j]$ de $(0, 0)$.

Hérédité sur i : Soit $i \in [\![0, n-1]\!]$ tel que pour tout $j \in [\![0, m]\!]$, (i, j) est distant de $\mathrm{M}[i][j]$ de $(0, 0)$.

Initialisation sur j : On suppose $j = 0$.
On peut montrer comme ci-dessus (pour les $(0, j)$) que la distance entre les sommets $(0, 0)$ et $(i+1, 0)$ est $\mathrm{M}[i+1][0] = i + 1$.

Hérédité sur j : Soit $j \in [\![0, m-1]\!]$ tel que la distance de $(i+1, j)$ à $(0, 0)$ vaut $\mathrm{M}[i+1][j]$.
Si `texte1[i]` $=$ `texte2[j]`, alors le sommet $(i+1, j+1)$ a trois prédécesseurs qui sont $(i, j+1), (i+1, j)$ et (i, j). Étant donné le poids des arêtes, la distance du sommet $(i+1, j+1)$ à $(0, 0)$ est alors le minimum de l'ensemble

$$\{\mathrm{M}[i][j], 1 + \mathrm{M}[i+1][j], 1 + \mathrm{M}[i][j+1]\}$$

Par construction de M, il s'agit bien de $\mathrm{M}[i][j] = \mathrm{M}[i+1][j+1]$.
Si `texte1[i]` $\neq$ `texte2[j]`, alors cette distance est bien

$$\min\{1 + \mathrm{M}[i+1][j], 1 + \mathrm{M}[i][j+1]\} = \mathrm{M}[i+1][j+1]$$

Conclusion sur j : Ainsi pour tout $j \in [\![0, m]\!]$, la distance de $(i+1, j)$ à $(0, 0)$ est $\mathrm{M}[i+1][j]$.

Conclusion sur i : Ainsi $\boxed{\text{pour tout } (i, j) \in [\![0, n]\!] \times [\![0, m]\!], \text{ la distance de } (i, j) \text{ à } (0, 0) \text{ est } \mathrm{M}[i][j]}$

Voici la fonction `successeurs(texte1, texte2, sommet)` :

```
def successeurs(texte1, texte2, sommet):
    n, m =len(texte1), len(texte2)
    i, j = sommet
    out =[]
    if i < n:
        out.append(((i+1, j), 1))
    if j < m:
        out.append(((i, j+1), 1))
        if i<n and texte1[i] == texte2[j]:
            out.append(((i+1, j+1), 0))
    return out
```

Question 15. L'algorithme de Dijkstra permet dans un graphe orienté pondéré de calculer la distance d'un sommet source à un sommet but.

Ici le sommet source est `origine` et le but est `sortie`. Or cette distance est $\mathrm{M}[n][m]$ qui la distance d'édition entre $\texttt{texte}_1$ et $\texttt{texte}_2$.

Les clés du dictionnaire `dist_final` renvoyé est l'ensemble des sommets visités lors de l'execution de la fonction. Cet ensemble est inclus dans la composante connexe du sommet `entrée`. Les valeurs dans `dist_final` sont les distances des clés au sommet `entrée` .

Question 16. Pour un graphe ayant S sommets et A arêtes,

l'algorithme `dijkstra` a une complexité de $\mathcal{O}((\mathrm{S}+\mathrm{A})\ln(\mathrm{S}))$.

Ici $\mathrm{A} \leqslant 3\mathrm{S}$ et $\mathrm{S} = (\texttt{len(texte}_1\texttt{)}+1) \times (\texttt{len(texte}_2\texttt{)}+1)$, on trouve donc une complexité en

$$\mathcal{O}\left(\texttt{len(texte}_1\texttt{)} \times \texttt{len(texte}_2\texttt{)} \times \ln\left(\texttt{len(texte}_1\texttt{)} \times \texttt{len(texte}_2\texttt{)}\right)\right)$$

On ne voit pas bien son intérêt par rapport à l'algorithme de programmation dynamique de la partie II.

Toutefois on ne calcule pas toutes les distances comme dans l'algorithme de la partie II.

Question 17. Je propose la fonction `h` qui a une complexité en $\mathcal{O}(1)$.

```
def h(texte1, texte2, sommet):
    n, m = len(texte1), len(texte2)
    i,j = sommet
    return n-i + m-j - 2*mini2(n-i,m-j)
```

Le parcours de (i,j) vers (n,m) comporte au maximum $\min(n-i, m-j)$ arêtes du types $(\ell,k) \to (\ell+1, k+1)$ de poids 0.

Il y a donc au moins $(n-i+m-j-2\min(n-i,m-j))$ arêtes de poids 1 empruntés pour aller du sommet (i,j) vers le sommet (n,m).

Ainsi `h(texte1, texte2, sommet)` est un minorant de la distance entre le sommet `sommet` et le sommet `sortie` du graphe.

La fonction `h` proposée est bien admissible.

Par Dijkstra, `dist_final` renvoyé est égal

```
{(0, 0) : 0, (0, 1) : 1, (1, 0) : 1, (2, 1) : 1, (0, 2) : 2, (1, 1) : 2, (2, 0) : 2,
             (2, 2) : 2, (3, 1) : 2, (1, 2) : 3, (3, 0) : 3, (3, 2) : 3}
```

alors qu'en utilisant l'algorithme A*, on trouve :

```
{(0,0): 0, (1,0): 1, (2,1) : 1, (0,1) : 1, (1,1) : 2, (2,0) : 2, (2,2) : 2, (3,1) : 2, (3,2) : 3}
```

Sur cet exemple, on voit que l'algorithme A* visite moins de sommets, ce qui permet de conjecturer qu'il a une meilleure complexité.