Gestion de versions de grans textes

X-ENS-ESPCI 2023

Sujet dans l'ensemble très difficile et de longueur délirante pour 2 heures. Faisable jusqu'à Q8, puis Q9 et Q10 difficiles (programmation dynamique, Levenshtein) peu guidée. Q11 extrêmement difficile et Q13 très difficile.

Q1.
```
def textes_egaux(texte1, texte2) :
# on suppose len(texte1) = len(texte2)
        for i in range(len(texte1)) :
            if texte1[i] != texte2[i] :
                return False
    return True
```

Q2.
```
def distance(texte1, texte2) :
    d = 0
    for i in range(len(texte1)) :
        if texte1[i] != texte2[i] :
            d += 1
    return d
```

Complexité est en $O(n)$.

Q3.
```
def aucun_caractere_commun(texte1, texte2) :
    # ces lignes équivalent à dico = {c : 0 for c in texte1}
    dico = {}
    for i in range(len(texte1)) :  # O(len(texte1)
         c = texte1[i]
         dico[c] = 0 # peu importe la valeur

    for j in range(len(texte2)) : # O(len(texte2))
        c = texte2[j]
        if c in dico :   # O(1) comme rappelé par l'énoncé
            return False

    return True
```

Q4. L'idée de la fonction est de détecter les débuts et les fins de tranches. Le préambule de l'énoncé autorise à manipuler des groupes d'éléments dans une liste avec la syntaxe `l[i:j]`. On n'est donc pas obligé de récupérer un à un les caractères de la tranche.

```
def differentiel(texte1, texte2) :
    liste_diff = []
    drapeau = False # signale qu'on est entré dans une zone différente

    for i in range(len(texte1)) :
        if not drapeau and texte1[i] != texte2[i] : # début de tranche
            ind_debut = i
            drapeau = True
        if drapeau and texte1[i] == texte2[i] : # fin de tranche
            drapeau = False
            tr = tranche(ind_debut, texte1[ind_debut:i], texte2[ind_debut:i])
            liste_diff.append(tr)
```

```
    # Attention au cas où on termine dans une tranche !
    if drapeau :
        tr = tranche(ind_debut, texte1[ind_debut:], texte2[ind_debut:])
        liste_diff.append(tr)
    return liste_diff
```

La ligne 5 itère `len(texte1)` fois, mais la ligne 11 n'est pas en $O(1)$ puisqu'elle manipule des groupes d'indice, elle est proportionnelle à la longueur de la tranche manipulée. Comme la somme des longueurs des tranches est inférieure à la longueur de la liste, toutes les itérations de la ligne 11 auront une complexité majorée par cette longueur. La fonction écrite ici a donc bien une complexité en $O((\text{len}(\text{texte1}))$.

Voici une autre version qui récupère les caractères un par un.

```
def differentiel_bis(texte1, texte2) :
    liste_diff = []
    drapeau = False # signale qu'on est entré dans une zone différente
    avant, apres = [], []
    for i in range(len(texte1)) :
        if texte1[i] != texte2[i] :
            avant.append(texte1[i])
            apres.append(texte2[i])
            if not drapeau : # début de tranche
                drapeau = True
                ind_debut = i
        else :
            if drapeau : # fin de tranche
                drapeau = False
                tr = tranche(ind_debut, avant, apres)
                liste_diff.append(tr)
                avant, apres  = [], []
    # Attention au cas où on termine dans une tranche !
    if drapeau :
        tr = tranche(ind_debut, avant, apres)
        liste_diff.append(tr)
    return liste_diff
```

Q5. Je propose à nouveau deux versions. La première modifie des groupes d'éléments dans une liste et je ne sais pas si c'est autorisé par l'énoncé. Dans le seconde, on modifie ces éléments un par un.

```
def applique(texte1, diff) :
    texte2 = texte1[:]
    for tr in diff :
        ind_deb = debut(tr)
        chaine_apres = apres(tr)
        ind_fin = fin(tr)
        texte2[ind_deb:ind_fin] = chaine_apres[:]
    return texte2
```

```
def applique_bis(texte1, diff) :
    texte2 = []
    for i in range(len(texte1)) : # je recopie d'abord texte1
        texte2.append(texte1[i])
    for tr in diff :
        ind_deb = debut(tr)
        chaine_apres = apres(tr)
```

```
        ind_fin = fin(tr)
        for i in range(ind_deb, ind_fin) :
            texte2[i] = chaine_apres[i-ind_deb]
    return texte2
```

Analysons la complexité de la seconde version. La ligne 2 est en $O(\text{len(texte1)})$. Dans la seconde boucle, on a une boucle interne ligne 9. La somme des longueurs des tranches est inférieure à la longueur de la liste, donc au total, la ligne 10 est exécutée au plus $len(texte1)$ fois. La complexité globale est donc bien en $O(\text{len(texte1)})$.

Q6. Il suffit de permuter `avant` et `apres` dans chaque tranche.

```
def inverse(diff) :
    inv_diff = []
    k = len(diff)
    for i in range(k) :
        tr = diff[i]                                                    5
        n_tr = tranche(debut(tr), apres(tr), avant(tr))
        inv_diff.append(n_tr)
    return inv_diff
```

Q7.
```
def modifie(texte_versionne, texte)     :
    remplace_courant(texte_versionne, texte)                            2
    texte1 = courant(texte_versionne)
    diff = differentiel(texte1, texte)
    historique(texte_versionne).append(diff)
```

```
def annule(texte_versionne) :
    diff = historique(texte_versionne).pop()
    inv_diff = inverse(diff)
    texte = courant(texte_versionne)
    texte_precedent = applique(texte, inv_diff)                         5
    remplace_courant(texte_versionne, texte_precedent)
    return texte_precedent
```

À cause des appels aux fonctions `diff`, `inverse` et `applique`, ces fonctions ont une complexité en $O(\text{len(texte1)})$.

Q8.
```
def poids(diff) :
    p = 0
    for i in range(len(diff)) :
        tr = diff[i]
        p += len(avant(tr))                                             5
        p += len(apres(tr))
    return p
```

Complexité en $O(\text{len(diff)})$.

Q9. — Si les deux textes présentent la même lettre aux indices i et j, elle n'a pas à subir de traitement pour passer de l'un à l'autre. Dans ce cas, il suffit de transformer `texte1[0:i]` en `texte2[0:j]` et le tour est joué pour `texte1[0:i+1]` et `texte2[0:j+1]`. On a donc $M[i+1][j+1] = M[i][j]$. Ce nombre de modifications élémentaires est minimal. En effet, si on pouvait transformer `texte1[0:i+1]` et `texte2[0:j+1]` en moins d'opérations, on pourrait aussi transformer `texte1[0:i]` et `texte2[0:j]` en moins d'opérations, et $M[i][j]$ ne serait pas un minimum.

— Sinon, c'est à dire si `texte1[i]` $\neq$ `texte2[j]`, il y a deux manières d'obtenir `texte2[0:j+1]` à partir

de `texte1[0:i+1]`.

— On passe de `texte1[0:i+1]` à `texte2[0:j]` (avec un coût $M[i+1][j]$) puis on ajoute le caractère `texte2[j]`.

— On supprime `texte1[i+1]` puis on passe de `texte1[0:i]` à `texte2[0:j+1]` (avec un coût $M[i][j+1]$).

Puisqu'on cherche un nombre d'étapes minimal, on retient la plus avantageuse de ces deux options. Dans ce cas

$$M[i+1][j+1] = 1 + \min(M[i+1][j], M[i][j+1])$$

À nouveau, il faudrait prouver par l'absurde qu'on obtient bien ainsi un nombre minimal d'étapes.

Q10. Attention. Soit $n = \text{len(texte1)}$ et $m = \text{len(texte2)}$. Le tableau M est de taille $(n+1) \times (m+1)$: en effet, pour $i = 0$, `texte1[0:i]` est le texte vide et le texte complet est `texte1[0:i+1]`. L'indication de l'énoncé est donc un peu trompeuse ! Je choisis une programmation dynamique montante. On commence par initialiser la première ligne et la première colonne qui sont triviales.

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

Complexité en $O(mn)$.

Q11. Je juge cette question 11 extrêmement difficile. La figure 3 nous permet de comprendre qu'il s'agit de parcourir la matrice M en partant de son coin inférieur droit. Les parties grisées horizontales correspondent aux portions « apres » à extraire de `texte2` (sauf la lettre la plus à gauche, par exemple XYZ) et les parties grisées verticales correspondent aux portions « avant » à extraire de `texte1` (sauf la lettre du haut, par exemple EF). Quand `texte1[i-1] == texte2[j-1]`, on est dans une zone commune aux deux textes et on remonte en diagonale. Sinon, on remonte vers le haut ou bien on se déplace vers la gauche en choisissant la plus petite des deux valeurs $M[i-1][j]$ et $M[i][j-1]$. Quand une zone commune se termine, on inscrit la tranche. Il faut encore parfois inscrire une tranche à la fin du parcours. Comme le parcours est fait à l'envers, il faut inverser les listes `avant` et `apres`, et la liste finale qui constitue le différentiel. C'est le rôle de la fonction auxiliaire `inverse`.

```
def inverse(L) :
    Linv = []
    n = len(L)
    for i in range(n) :
        Linv.append(L[n-1-i])
    return Linv

inverse('1234')

def différentiel(texte1, texte2, M) :
    diff = []
    i, j = len(texte1), len(texte2)
```

```
    avant, apres = [], []

    while (i, j) != (0, 0) :
        if i == 0 : # on est bloqué en haut,
            avant.append(texte2[j]) # on collecte la lettre texte2
            j -= 1 # on va vers la gauche
        elif j == 0 :  # on est bloqué à gauche
            apres.append(texte1[i]) # on collecte la lettre de texte1
            i -= 1 # on va vers le haut

        else : # ni i ni j n'est nul, on ne touche pas les bords
            if texte1[i-1] == texte2[j-1] : # on découvre une zone commune
                if avant != [] or apres != [] :
                    avant = inverse(avant)
                    apres = inverse(apres)
                    tr = tranche(i, avant, j, apres)
                    diff.append(tr)
                i -= 1
                j -= 1
                avant = [] # on réinitialise pour la prochaine tranche
                apres = []

            else :
                if M[i][j-1] <= M[i-1][j] :
                    j -= 1  # on part vers la gauche
                    apres.append(texte2[j]) #  collecte la lettre de texte2
                else :
                    i -= 1  # on part vers le haut
                    avant.append(texte1[i]) #  collecte la lettre de texte1
    ## Parcours terminé
    if avant != [] or apres != [] :
        avant = inverse(avant)
        apres = inverse(apres)
        tr = tranche(i, avant, j, apres)
        diff.append(tr)

    diff = inverse(diff)
    return diff
```

Q12. Considérons deux intervalles $[d_1, f_1[$ et $[d_2, f_2[$ (je prends des intervalles ouverts à droite car en Python, l'indice supérieur est exclu). Il sont disjoints si

$$f_1 \leqslant d_2 \quad \text{ou} \quad f_2 \leqslant d_1 \quad .$$

Il sont en conflit si cette condition est enfreinte, c'est à dire si

$$f_1 > d_2 \quad \text{et} \quad f_2 > d_1 \quad .$$

On pourrait tester les conflits entre tous les couples de tranches prises dans `diff1` et dans `diff2`, mais on aurait alors une complexité en $O(\text{len(diff1)} \times \text{len(diff2)})$ et cela ne répondrait pas à la question. On profite du fait que les tranches sont ordonnées dans chaque différentiel. Quand deux tranches ne sont pas en conflit, celle qui termine la première ne peut pas être en conflit avec toutes celles qui se trouvent après l'autre dans le différentiel auquel cette autre appartient. Il est donc inutile d'examiner ses conflits éventuels avec ces tranches ultérieures, et on peut passer à la suivante.

```
def conflit_tranches(tr1, tr2) :
    d1, f1 = debut_avant(tr1), fin_avant(tr1)
    d2, f2 = debut_avant(tr2), fin_avant(tr2)
    return f1 > d2 and f2 > d1

def conflit(diff1, diff2) :
    i, j = 0, 0
    while i < len(diff1) and j < len(diff2) :
        tr1 , tr2 = diff1[i], diff2[j]
        if conflit_tranches(tr1, tr2) :
            return True
        fin1 = fin_avant(tr1)
        fin2 = fin_avant(tr2)
        if fin1 <= fin2 :
            i += 1
        else :
            j += 1
    return False
```

À chaque itération, i ou j est incrémenté donc $i+j$ augmente de 1. On $i+j$ ne peut pas excéder `len(diff1)+ len(diff2)`. Donc la complexité est en $O(\text{len(}\texttt{diff1}\text{)} + \text{len(}\texttt{diff2}\text{)})$.

Q13. Cette question me paraît très difficile.

Les arguments `avant` et `apres` des tranches de `diff2` expriment les modifications du texte apportées par le second auteur et il ne faut pas les changer. Cette idée est cohérente avec l'affirmation de l'énoncé : `poids(fusionne(diff1, diff2))= poids(diff2)`.

Par contre, l'application du différentiel `diff1` au texte décale les indices des caractères dans le texte, de sorte que les emplacements concernés par les modification du second auteur sont eux aussi décalés. Plus précisément, l'écriture d'une tranche `tr1` de `diff1` entraîne un décalage d'indices égal à `len(apres(tr1) - len(avant(tr1))`, et ce pour toutes les tranches de `diff2` qui se trouvent à sa droite. On doit donc, pour ces tranches de `diff2`, modifier les valeurs associées aux clés `'debut_avant'` et `'debut_apres'`. Chaque tranche de `diff2` va donc subir le décalage cumulé de toutes les tranches de `diff1` qui se trouvent à sa gauche.

```
def decalage_tranche(tr, valeur_decal) :
''' renvoie une nouvelle tranche en décalant les indices'''
    d_av = debut_avant(tr)
    d_ap = debut_apres(tr)
    return tranche(d_av + valeur_decal, avant(tr), a_ap + valeur_decal,
    apres(tr)
```

```
def fusionne(diff1, diff2) :
    n_diff = []
    n1, n2 = len(diff1), len(diff2)
    i1, i2 = 0, 0
    valeur_decalage = 0
    while i2 < n2 :
        tr2 = diff[i2]
        tr1 = diff[i1]
        # on cherche les tranche de diff1 qui se trouvent à gauhe de tr2
        while i1 < n1 and fin_avant(tr1) < debut_avant(tr2) :
            valeur_decalage += len(apres[tr1]) - len(avant[tr1])
            i1 += 1
        n_tr = decalage_tranche(tr2, valeur_decalage)
```

```
            n_diff.append(n_tr)
            i2 += 1
        return n_diff
```

Q14. Ce graphe exprime l'idée qui sous-tend la programmation dynamique pour le calcul de la distance de Lewenshtein. Pour aller de (i, j) et $(i+1, j)$ ou vers $(i, j+1)$, on fait une opération élémentaire de plus donc on attribue la valeur 1 à l'arrête. On ne peut se déplacer en diagonale de (i, j) vers $(i+1, j+1)$ que si `texte1[i] == texte2[j]`, et dans ce cas l'arête possède un poids nul.

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

Je renonce à la preuve par récurrence demandée. Prendre le plus court chemin depuis (0,0) dans ce graphe revient à appliquer la méthode de programmation dynamique de la distance de Levenshtein.

Q15. On reconnaît dans le code de la figure 5 les grandes étapes de l'algorithme de Dijsktra. Tout d'abord, la distance est initialisée à 0 pour le point de départ, puis on itère les deux actions suivantes : sélection d'un sommet de distance minimale, révision des étiquettes. Progressivement, on étend l'ensemble des nœuds pour lesquels la distance (c'est à dire le plus court chemin depuis l'origine) est connue, et on cesse dès que la sortie, c'est à dire le nœud décrivant les deux textes en entier, est atteinte. Dans le dictionnaire `dist_final` renvoyé,

— les clés sont les sommets du graphe, correspondant à des fractions de `texte1` et `texte2` ;

— les valeurs sont les plus courts chemins de l'origine à ces fractions de texte, c'est à dire les distances de Levenshtein. La distance d'édition entre `texte1` et `texte2` est la valeur associée à la clé `((len(texte1), len(texte2))`.

Q16. Soit A le nombre d'arêtes du graphe et $|S|$ son nombre de sommets. En utilisant un file de priorité, on donne à l'algorithme de Dijkstra une complexité en $O(A+S)\log S$. Dans notre cas,

— $S \simeq \text{len(texte1)} \times \text{len(texte2)}$;

— $A \simeq 3S$

donc la complexité est en $O(P \ln P)$ avec $P = \text{len(texte1)} \times \text{len(texte2)}$. La complexité est donc moins avantageuse que dans la partie précédente. Il est cependant à noter que les itérations peuvent s'arrêter avant qu'on ait parcouru tout le graphe. Dans certaines situations favorables, il se peut que le calcul soit plus rapide qu'en remplissant toute la matrice de distance d'édition