Gestion de versions de grands textes

I Différentiels par positions fixes

Dans toute cette partie, on suppose que les textes comparés ont la même taille.

Q1 La fonction `textes_égaux` teste caractère par caractère que les listes de caractères `texte1` et `texte2` sont égales.

```python
def textes_égaux(texte1, texte2):
    n = len(texte1) # hypothèse de l'énoncé : len(texte1) == len(texte2)
    for i in range(n):
        if texte1[i] != texte2[i]:
            return False
    return True
```

La complexité de la fonction `textes_egaux` est en $\boxed{\mathcal{O}(n)}$ où n est la longueur commune des deux textes.

Q2 La fonction `distance` calcule ce que l'on appelle la distance de Hamming entre les listes de caractères `texte1` et `texte2`.

```python
def distance(texte1, texte2):
    n = len(texte1) # hypothèse de l'énoncé : len(texte1) == len(texte2)
    compteur = 0
    for i in range(n):
        if texte1[i] != texte2[i]:
            compteur += 1
    return compteur
```

La complexité de la fonction `distance` est en $\boxed{\mathcal{O}(n)}$ où n est la longueur commune des deux textes.

Q3 On suppose dans cette question que les listes de caractères `texte1` et `texte2` peuvent être de longueurs différentes. *Dans le cas où `texte1` et `texte2` sont toutes deux des listes vides, la fonction renvoie `True`.*

```python
def aucun_caractère_commun(texte1, texte2):
    n1, n2 = len(texte1), len(texte2)
    dico1 = {texte1[i]:True for i in range(n1)} # dictionnaire des caractères de texte1
    for i in range(n2):
        if texte2[i] in dico1:
            return False
    return True
```

Complexité de la fonction `aucun_caractère_commun` :

La complexité de la fonction dépend de `len(t1)` et de `len(texte2)`.

- À l'extérieur de la boucle **`for`**, les instructions sont en $\mathcal{O}(1)$, à l'exception de la création du dictionnaire `dico1` en $\mathcal{O}($`len(texte1)`$)$.
- À chacun des `len(texte2)` tours de boucle, les instructions exécutées, en particulier le test `texte2[i] in dico1`, sont en $\mathcal{O}(1)$.
  La boucle **`for`** a donc une complexité en $\mathcal{O}($`len(texte2)`$)$.

Par somme, la complexité de la fonction `aucun_caractère_commun` est bien en $\mathcal{O}($`len(texte1) + len(texte2)`$)$, comme demandé par l'énoncé.

Q4

```python
def différentiel(texte1, texte2):
    n = len(texte1) # hypothèse de l'énoncé : len(texte1) == len(texte2)
    diff = [] # différentiel de texte2 vis-à-vis de texte1

    i = 0     # indice de parcours de texte1
    while i < n:
        while i < n and texte1[i] == texte2[i]: # on avance jusqu'au premier caractère différent, s'il existe
            i += 1

        # ici, i == n ou texte1[i] != texte2[i]
        arg_début, arg_avant, arg_après = i, [], []
        while i < n and texte1[i] != texte2[i]: # on avance jusqu'au dernier caractère différent et on complète arg_avant et arg_après
            arg_avant.append(texte1[i])
            arg_après.append(texte2[i])
            i += 1

        # ici, i == n ou texte1[i] == texte2[i]
        if arg_avant != [] : # si arg_avant (et donc arg_après) a été alimenté, on crée une tranche
            tr = tranche(arg_début, arg_avant, arg_après)
            diff.append(tr)

    return diff
```

Complexité de la fonction `differentiel` :

La complexité de la fonction dépend de n = `len(texte1)` = `len(texte2)`.

— À l'extérieur de la boucle **`while`** principale, les instructions sont en $\mathcal{O}(1)$.

— À chacun des n tours de boucle, les instructions exécutées sont en $\mathcal{O}(1)$.
En effet, on réalise une première boucle **`while`** avec un indice `i` qui est incrémenté et vaut au plus $n-1$ et une seconde boucle **`while`** qui démarre avec la dernière valeur de l'indice `i`. Par ailleurs, toutes les instructions sont en $\mathcal{O}(1)$, y compris l'exécution de la fonction `tranche`.
La boucle **`while`** principale a donc une complexité en $\mathcal{O}(n)$.

Par somme, la complexité de la fonction `differentiel` est en $\boxed{\mathcal{O}(n) = \mathcal{O}(\texttt{len(texte1)})}$, où n est la longueur commune des deux textes.

Q5 On crée une fonction auxiliaire pour concaténer des listes en se limitant à l'utilisation de `append`.

```python
def ajout(texte1, texte2):
    '''renvoie texte1 + texte2.'''
    for i in range(len(texte2)):
        texte1.append(texte2[i])
```

Dans la fonction `applique` :

— soit `diff` est vide, et on peut renvoyer directement `texte1`,

— soit `diff` n'est pas vide, et on construit `texte2` à partir d'une liste vide en lui ajoutant ses caractères un à un (ou par slicing) en utilisant les tranches données (dont le nombre est majoré par $\lceil \frac{n}{2} \rceil$, car chaque tranche est séparée d'au moins 1 caractère). Ceci garantit une complexité en $\boxed{\mathcal{O}(n)}$, où n est la longueur commune des deux textes.

```python
def applique(texte1, diff):
    if diff == []:
        return texte1[:] # copie de texte1
    else: # diff contient des tranches à modifier
        texte2 = [] # on construit texte2 en partant de la liste vide

        arg_début = 0
        for tr in diff: # pour chaque tranche: d'une part, on récupère les caractères identiques précédents (s'ils existent), d'autre part, on ajoute les caractères donnés par la tranche
            ajout(texte2, texte1[arg_début: début(tr)]) # caractères non modifiés
            ajout(texte2, après(tr)) # caractères donnés par la tranche
            arg_début = fin(tr)

        ajout(texte2, texte1[arg_début:]) # caractères non modifiés éventuels après la dernière tranche
        return texte2
```

Q6 Pour répondre aux contraintes données par l'énoncé, il suffit d'intervertir les textes avant et après dans chaque tranche de `diff` dans une nouvelle version de `diff`.

```python
def inverse(diff):
    diff_inv = []
    for tr in diff:
        diff_inv.append(tranche(début(tr), après(tr), avant(tr)))
    return diff_inv
```

La complexité de la fonction `inverse` est en $\boxed{\mathcal{O}(\texttt{len(diff)})}$.

Q7 La procédure `modifie(texte_versionné, texte)` :

- détermine le différentiel de `texte` vis-à-vis du texte courant contenu dans le dictionnaire `texte_versionné` (l'énoncé dit de faire l'hypothèse que `texte` a la même taille que le texte courant),
- empile ce nouveau différentiel dans l'historique des différentiels,
- remplace le texte courant par `texte`.

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

La complexité de la fonction `modifie` est en $\boxed{\mathcal{O}(\texttt{len(diff)})}$.
Comme `len(diff)` a une taille au plus égale à $\lceil \frac{n}{2} \rceil$ (comme vu en **Q5**), la complexité de la fonction `modifie` est aussi en $\boxed{\mathcal{O}(n)}$, où n est la longueur de `texte`.

La fonction `annule(texte_versionné)` :

- dépile le dernier différentiel de l'historique de `texte_versionné`,
- récupère le texte courant de `texte_versionné`,
- détermine le texte précédent en appliquant l'inverse du dernier différentiel au texte courant,
- remplace le texte courant par le texte précédent,
- renvoie le texte précédent.

L'énoncé indique que la fonction `annule` n'est jamais appelée quand il n'y a pas eu de modification. On ne demande pas de programmation défensive.

```python
def annule(texte_versionné):
    dernier_diff = historique(texte_versionné).pop()
    texte2 = courant(texte_versionné)
    texte1 = applique(texte2, inverse(dernier_diff))
    remplace_courant(texte_versionné, texte1)
    return texte1
```

La complexité de la fonction `annule` est en $\boxed{\mathcal{O}(n)}$ où n `= len(texte)`.

II Différentiels sur des positions variables

Q8

```python
def poids(diff):
    weight = 0
    for tr in diff:
        weight += len(avant(tr)) + len(après(tr))
    return weight
```

La complexité de la fonction `poids` est en $\boxed{\mathcal{O}(\texttt{len(diff)})}$.

Q9 Soit $0 \leqslant i \leqslant$`len(texte1)` et $0 \leqslant j \leqslant$`len(texte2)`.
Soit `M[i][j]`, le nombre minimal de suppressions et d'insertions de caractères pour passer de `texte1[:i]` à `texte1[:j]`.

Une relation de récurrence est donnée par,

si `texte1[i] = texte1[j]`,
- alors `M[i+1][j+1] = M[i][j]`
- sinon `M[i+1][j+1] = min(M[i+1][j], M[i][j+1]) + 1`

Explication de la relation de récurrence :

Soit `M[i][j]` donné pour passer de `texte1[:i]` à `texte2[:j]`. Si `texte1[i]` et `texte2[j]` sont **égaux**, alors on n'ajoute ni supprime de caractère pour passer de `texte1[:i+1]` à `texte2[:j+1]`. Dans ce cas, le coût d'édition de `M[i+1][j+1]` est égal à `M[i][j]`.

Si `texte1[i]` et `texte2[j]` sont **différents**, alors on a deux possibilités pour obtenir le nombre minimal de suppressions et insertions pour passer de `texte1[:i+1]` à `texte2[:j+1]` :

— `M[i+1][j]` est le nombre minimal de suppressions et insertions pour passer de `texte1[:i+1]` à `texte2[:j]`. En ajoutant `texte2[j]` à `texte2`, on ajoute 1 à `M[i+1][j]` pour obtenir le nombre minimal de suppressions et insertions pour passer de `texte1[:i+1]` à `texte2[:j+1]`. Donc, `M[i+1][j+1]` vaut `M[i+1][j] + 1`.

— `M[i][j+1]` est le nombre minimal de suppressions et insertions pour passer de `texte1[:i]` à `texte2[:j+1]`. En supprimant `texte1[i]` de `texte1`, on ajoute 1 à `M[i][j+1]` pour obtenir le nombre minimal de suppressions et insertions pour passer de `texte1[:i+1]` à `texte2[:j+1]`. Donc, `M[i+1][j+1]` vaut `M[i][j+1] + 1`.

Par définition, `M[i+1][j+1]` et la plus petite de ces deux valeurs.

Q10 On utilise une programmation dynamique **ascendante** afin de générer la matrice de distance d'édition,

```python
def mini(a, b):
    '''renvoie le plus petit des deux nombres a et b.'''
    if a < b: return a
    return b
```

```python
def levenshtein(texte1, texte2):
    n, m = len(texte1), len(texte2)
    M = [[0 for j in range(m + 1)] for i in range(n + 1)] # structure de mémoïsation
    for i in range(n + 1):
        for j in range(m + 1):
            if i == 0 : M[i][j] = j
            elif j == 0 : M[i][j] = i
            else: # i > 0 et j > 0
                if texte1[i-1] == texte2[j-1] :
                    M[i][j] = M[i-1][j-1]
                else:
                    M[i][j] = mini(M[i-1][j], M[i][j-1]) + 1
    return M
```

La complexité de la fonction `levenshtein` est en $\boxed{\mathcal{O}(\texttt{len(texte1)}\times\texttt{len(texte2)})}$.
Elle est donc bien polynomiale en n = `max(len(texte1), len(texte2))`.

Q11 On utilise la fonction auxiliaire `inv` qui inverse les termes d'une liste et la fonction `concaténation(texte1, texte2)` qui renvoie `texte1 + texte 2`.

```python
def inv(liste):
    ''' inverse les termes d'une liste.'''
    n = len(liste)
    inv_liste = []
    for i in range(n):
        inv_liste.append(liste[n-1-i])
    return inv_liste

def concaténation(texte1, texte2):
    '''on concatène texte1 avec texte2.'''
    arg = []
    ajout(arg, texte1)
    ajout(arg, texte2)
    return arg
```

La fonction `differentiel` parcourt la matrice des distances de Levenshtein en partant de « la fin » (i.e. `M[len(texte1)][len(texte2)]`) et en remontant jusqu'au début (i.e. `M[0)][0]`). Elle crée progressivement les tranches du différentiel en démarrant par la dernière tranche (d'où l'utilisation de la fonction `inv` pour inverser in fine les tranches du différentiel).

Pour tout $0 < i \leqslant$`len(texte1)` et $0 < j \leqslant$`len(texte2)`,

- si `texte1[i-1]` est différent de `texte2[j-1]`,
  - si `M[i][j-1] <= M[i-1][j]`, alors on alimente `inv_arg_après` (i.e. `arg_après` en sens inverse) avec `texte2[j-1]`. En cas d'égalité, on fait également ce choix, cf. Figure 3 de l'énoncé,
  - sinon `M[i][j-1] > M[i-1][j]`, alors on alimente `inv_arg_avant` (i.e. `arg_avant` en sens inverse) avec `texte1[i-1]`.
- si `texte1[i-1]` est égal à `texte2[j-1]`, on insère une nouvelle tranche dans le différentiel à condition que `inv_arg_avant` ou `inv_arg_après` soit non vide : ceci

permet entre-autres de n'ajouter la tranche qu'une seule fois si l'égalité de caractères se poursuit. Lors de l'insertion de la tranche, on doit veiller à ce que les éléments de `arg_avant` et `arg_après` soient dans le sens de lecture (d'où l'utilisation de la fonction `inv` pour l'inversion).

Une fois arrivé au bord, on complète `arg_avant` ou `arg_après` de la tranche respectivement par `texte2[:j]` (quand `i` est nul) ou `texte1[:i]` (quand `j` est nul).

```
def différentiel(texte1, texte2, M):
    n1, n2 = len(texte1), len(texte2)
    i, j = n1, n2

    diff = []
    inv_arg_avant, inv_arg_après = [], [] # la collecte des lettres est inversée car on part de la fin
    while i != 0 or j != 0:
        if i == 0 : # bord haut, on complète inv_arg_après par texte2[:j]
            arg_après = concaténation(texte2[:j], inv(inv_arg_après))
            diff.append(tranche(i, inv(inv_arg_avant), 0, arg_après))
            j = 0
        elif j == 0 : # bord gauche, on complète inv_arg_avant par texte1[:i]
            arg_avant = concaténation(texte1[:i], inv(inv_arg_avant))
            diff.append(tranche(0, arg_avant, j, inv(inv_arg_après)))
            i = 0
        else: # i > 0 et j > 0
            if texte1[i-1] == texte2[j-1] : # on remonte en diagonale
                if inv_arg_avant != [] or inv_arg_après != []: # on crée la tranche une seule fois dès qu'elle est non vide
                    diff.append(tranche(i, inv(inv_arg_avant), j, inv(inv_arg_après)))
                    inv_arg_avant, inv_arg_après = [], []
                i, j = i-1, j-1
            else: # texte1[i-1] != texte2[j-1]
                if M[i-1][j] < M[i][j-1]: # on monte d'une ligne et on complète inv_arg_avant
                    inv_arg_avant.append(texte1[i-1])
                    i -= 1
                else: # M[i][j-1] <= M[i-1][j], on se décale à gauche et on complète inv_arg_après
                    inv_arg_après.append(texte2[j-1])
                    j -= 1

    return inv(diff) # la collecte des tranches est inversée car on part de la fin
```

<u>Explication de la complexité :</u>

La complexité de la fonction `différentiel` est en $\boxed{\mathcal{O}(\texttt{len(texte1)}+\texttt{len(texte2)})}$.

En effet, au sein de la boucle `while`, à chaque test, on décrémente l'indice `i` ou `j` (ou les deux). Ainsi la boucle fait exactement `len(texte1)+len(texte2)` tours.

Concernant la fonction `inv`,

— elle permute au plus une seule fois chaque caractère de `texte1` et `texte2`, ce qui se déroule en $\mathcal{O}(\texttt{len(texte1)}+\texttt{len(texte2)})$,

— elle inverse une seule fois les `len(diff)` tranches de `diff`, avec `len(diff)` qui est un $\mathcal{O}(\texttt{len(texte1)}+\texttt{len(texte2)})$.

Concernant la fonction `concaténation`, elle n'est utilisée qu'une seule fois lors de l'arrivée au bord.

- Pour $i = 0$, la complexité du slicing `texte2[:j]` est en $\mathcal{O}(j)$; à ce texte slicé, de taille au plus égale à `len(texte2)`, est concaténé l'inverse de `inv_arg_après`, de taille au plus égale à `len(texte2)`. La complexité de l'instruction `arg_après = concaténation(texte2[:j], inv(inv_arg_après))` est donc en $\mathcal{O}($`len(texte2)`$)$.
- Pour $j = 0$, de façon symétrique, la complexité de l'instruction `arg_avant = concaténation(texte1[:i], inv(inv_arg_avant))` est en $\mathcal{O}($`len(texte1)`$)$.

D'où le résultat sur la complexité.

Explication des propriétés du différentiel fournit par la fonction `différentiel` :

Par construction,

- Comme `i` et `j` correspondent respectivement à `arg_début_avant` et `arg_début_après` d'une tranche, et sont décrémentés (l'un ou les deux) lors de la constitution d'une tranche, les suites des `arg_début_avant` et `arg_début_après` sont croissantes.
- On ajoute un caractère à `arg_avant` ou `arg_après` uniquement s'ils sont différents. L'indice `arg_début_avant`, respectivement, `arg_début_après` ne change pas à l'ajout d'un caractère respectivement dans `arg_après` et `arg_avant` et, dès qu'un caractère pourrait être en commun, on ajoute une tranche : ceci fait que l'on n'a aucun caractère en commun à `arg_avant` et `arg_après`.
- On ajoute systématiquement un caractère à `arg_avant` ou `arg_après` à chaque tour de boucle, sauf si le caractère est le même. Si tous les caractères sont les mêmes, `diff` reste vide. Ainsi, l'une des deux listes `arg_avant` ou `arg_après` est toujours non vide.

On a donc bien les propriétés d'un différentiel.

Q12 L'algorithme suivant exploite le fait que les dates de début et fin des tranches sont globalement croissantes (voire strictement entre `fin_avant` d'une tranche et `début_avant` de la tranche suivante).

```python
def conflit(diff1, diff2):
    m1, m2 = len(diff1), len(diff2)
    i, j = 0, 0
    while i < m1 and j < m2:
        tr1, tr2 = diff1[i], diff2[j]
        if (fin_avant(tr2) >= début_avant(tr1)) and (début_avant(tr2) <=
    fin_avant(tr1)): # existence d'un conflit
            return True
        else:
            if début_avant(tr2) >= début_avant(tr1):
                i += 1
            else:
                j += 1

    return False # pas de conflit détecté
```

Complexité de la fonction `conflit` :

La complexité de la fonction dépend de $m1$ = `len(diff1)` et de $m2$ = `len(diff2)`.

— À l'extérieur de la boucle **`while`**, les instructions sont en $\mathcal{O}(1)$.

— Dans le pire des cas, on exécute $(m1-1)+m2$ (ou $(m2-1)+m1$ par symétrie) tours de boucle. Les instructions exécutées sont en $\mathcal{O}(1)$. La boucle **`while`** a donc une complexité en $\mathcal{O}(m1+m2)$.

Par somme, la complexité de la fonction `conflit` est donc bien en
$\boxed{\mathcal{O}(\texttt{len(diff1) + len(diff2)})}$, comme demandé par l'énoncé.

Q13 L'application d'une tranche de `diff2` à gauche d'une tranche de `diff1` sur le texte `texte` (tel qu'exposé dans l'énoncé) ne pose aucune difficulté : cela revient à les appliquer l'une à la suite de l'autre, à partir de `texte` (i.e. `fusionne(diff1, diff2)` renvoie `diff2`).
Toute la difficulté consiste à faire l'inverse : appliquer des tranches de `diff2` à droite de tranches de `diff1` ; il faut alors décaler en conséquence les positions `début_avant(tr2)` et `début_après(tr2)` : on utilise alors la fonction auxiliaire `applique_ecart_de_position`.

```
def applique_ecart_de_position(tr, écart_position):
    ''' décale les positions début_avant(tr) et début_après(tr) de tr de écart_position.'''
    return tranche(début_avant(tr) + écart_position, avant(tr), début_après(tr) + écart_position, après(tr))
```

On fusionne des différentiels sur des positions variables. On a vérifié auparavant qu'ils sont sans conflit.

L'algorithme suivant commence par traiter le cas où `diff1` ou `diff2` sont vides.

Dans le cas où ce n'est pas le cas, `diff1` et `diff2` contiennent tous les deux au moins un « groupe » de tranches (éventuellement réduit à 1 tranche) que nous considérons dans l'ordre suivant :
Groupe de `diff2` (éventuellement vide) - Groupe de `diff1` - Groupe de `diff2` (éventuellement vide si le premier Groupe de `diff2` ne l'est pas) -...- Groupe de `diff1` (éventuellement vide).

Nommons `diff`, initialement vide, le résultat de la fusion de `diff1` et `diff2`.

— On commence par vérifier s'il existe un groupe de tranches de `diff2` qui précède un groupe de tranches de `diff1`. En effet, si tel est le cas, aucun écart de position n'est à effectuer, et le groupe de tranches de `diff2` est récupéré tel quel dans `diff`.

— À ce stade, on est assuré qu'il existe un groupe de tranches de `diff1`, éventuellement suivi d'un groupe de tranches de `diff2`.

  On itère sur les groupes de tranches de `diff2` ce qui suit :

  - Soit il n'y a plus de groupe de tranches de `diff2` et le traitement s'arrête en renvoyant `diff`.
  - Sinon, on est assuré qu'il existe un groupe de tranches de `diff1`, suivi d'un groupe de tranches de `diff2`.

    On parcourt le groupe de tranches de `diff1` afin de calculer l'écart de position à apporter au groupe de tranches de `diff2` situé « à sa droite » (écart « cumulé » lors d'itérations).

    Puis, on parcourt le groupe de tranches de `diff2` pour lui appliquer l'écart de position avec la fonction `applique_ecart_de_position`. Les nouvelles tranches portant les modifications sont récupérées dans `diff`.

- On a alors plusieurs cas :
  - Soit il s'agissait du dernier groupe de tranches de `diff2` et le traitement s'arrête en renvoyant `diff`.
  - Soit il reste au moins un groupe de tranches de `diff2`. Cela signifie qu'il y a forcément un groupe de tranches de `diff1` qui le précède car les différentiels sont sans conflit. Ceci permet d'itérer.

```python
def fusionne(diff1, diff2):
    m1, m2 = len(diff1), len(diff2)

    if diff1 == [] : return diff2 # on applique diff2 sur le texte
    elif diff2 == []: return [] # on conserve les modifs de diff1

    # ici, diff1 et diff2 sont non vides
    diff = [] # création d'un nouveau différentiel
    i, j = 0, 0

    # il existe un groupe de tranche de diff2 qui précède un groupe de tranches de diff1
    while j < m2 and début_avant(diff1[i]) > début_après(diff2[j]):
        diff.append(diff2[j])
        j += 1

    # il existe un groupe de tranches de diff1, éventuellement suivi d'un groupe de tranches de diff2
    écart_position =  0
    # s'il n'y a plus de groupe de tranches de diff2, le traitement s'arrête en renvoyant diff
    while j < m2:
        # on calcule l'écart de position lié aux tranches du groupe de diff1
        while i < m1 and début_après(diff1[i]) < début_avant(diff2[j]):
            écart_position += len(après(diff1[i])) - len(avant(diff1[i]))
            i += 1

        # on applique l'écart de position à toutes les tranches du groupe de diff2  mises dans diff
        while j < m2 :
            tr = applique_ecart_de_position(diff2[j], écart_position)
            diff.append(tr)
            j += 1

    return diff
```

On parcourt itérativement les tranches de `diff1` et `diff2`, en n'utilisant que des instructions en $\mathcal{O}(1)$, donc la complexité de la fonction `fusionne` est bien en $\boxed{\mathcal{O}(\texttt{len(diff1)}+\texttt{len(diff2)})}$.

III Calcul de différentiels par calcul de plus courts chemins

Q14

```python
def successeurs(texte1, texte2, sommet):
    n1, n2 = len(texte1), len(texte2)
    i, j = sommet
    succ = []
    if i < n1:
        succ.append(((i+1, j), 1))
    if j < n2:
        succ.append(((i, j+1), 1))
    if i < n1 and j < n2 and texte1[i] == texte2[j]:
        succ.append(((i+1, j+1), 0))
    return succ
```

Montrons par récurrence que, pour tout $0 \leqslant i \leqslant$`len(texte1)` et $0 \leqslant j \leqslant$`len(texte2)`, la longueur d'un plus court chemin de $(0,0)$ à (i,j) est égal à `M[i][j]` avec la pondération utilisée pour les arcs du graphe :
On fait une récurrence double sur $0 \leqslant i \leqslant$`len(texte1)` et $0 \leqslant j \leqslant$`len(texte2)`.

— Soit $i = 0$ et $j \geqslant 0$: il existe un unique chemin menant de $(0,0)$ à (i,j) : il s'agit donc du plus court chemin à partir de $(0,0)$. Comme sa longueur est j, elle est bien égale à `M[0][j]`.

— Soit $0 \leqslant i <$`len(texte1)`. On suppose que pour tout $0 \leqslant j \leqslant$`len(texte2)`, la longueur d'un plus court chemin de $(0,0)$ à (i,j) est égale à `M[i][j]` (HR1).

- Soit $j = 0$: il existe un unique chemin menant de $(0,0)$ à $(i+1,0)$: il s'agit donc du plus court chemin à partir de $(0,0)$. Comme sa longueur est $i+1$, elle est bien égale à `M[i+1][0]`.
- Soit $0 \leqslant j <$`len(texte2)` : on suppose que la longueur d'un plus court chemin de $(0,0)$ à $(i+1,j)$ est égale à `M[i+1][j]` (HR2).
Un plus court chemin pour aller de $(0,0)$ à $(i+1,j+1)$ passe par (i,j) ou $(i+1,j)$ ou $(i,j+1)$.
    - Si le chemin passe par (i,j), il existe un arc $(i,j) \to (i+1,j+1)$. Dans ce cas, un plus court chemin a pour poids celui d'un chemin le plus court passant par (i,j), de poids `M[i][j]` (hypothèse de récurrence (HR1)), auquel on ajoute le poids de l'arc $(i,j) \to (i+1,j+1)$, à savoir 0. Dans ce cas, un plus court chemin a pour poids `M[i][j]`.
    - Si le chemin passe par $(i+1,j)$, le chemin a pour poids celui d'un chemin le plus court passant par $(i+1,j)$, de poids `M[i+1][j]` (hypothèse de récurrence (HR2)), auquel on ajoute le poids de l'arc $(i+1,j) \to (i+1,j+1)$, à savoir 1. Dans ce cas, un plus court chemin a pour poids `M[i+1][j] + 1`.
    - Si le chemin passe par $(i,j+1)$, le chemin a pour poids celui d'un chemin le plus court passant par $(i,j+1)$, de poids `M[i][j+1]` (hypothèse de récurrence (HR1)), auquel on ajoute le poids de l'arc $(i,j+1) \to (i+1,j+1)$, à savoir 1. Dans ce cas, un plus court chemin a pour poids `M[i][j+1] + 1`.

Notons que dans le premier cas, on ne supprime, ni n'ajoute de caractère, càd. `texte1[i]` est égal à `texte2[j]` et dans les 2 derniers cas, on ajoute un caractère à `texte1` ou (exclusif) à `texte2`, donc `texte1[i]` et `texte2[j]` sont différents. On retrouve bien que le poids d'un plus court chemin allant de $(0,0)$ à $(i+1,j+1)$ est `M[i+1][j+1]` (par définition des éléments de la matrice de Levenshtein, cf. question 9). Ceci achève la récurrence.

Q15 L'algorithme de Dijkstra permet de calculer le plus court chemin du sommet d'entrée $(0,0)$ à tout sommet (i,j) du graphe, donc avec $0 \leqslant i \leqslant$`len(texte1)` et $0 \leqslant j \leqslant$`len(texte2)`. Or, nous venons de prouver qu'il s'agit aussi de la distance d'édition entre `texte1[:i]` et `texte2[:j]`.
En particulier, lorsque l'algorithme s'arrête, il est arrivé au sommet de sortie, de coordonnées `(len(texte1), len(texte2))`. Il donne alors la distance d'un chemin le plus court entre le sommet d'entrée et le sommet de sortie, càd. la distance d'édition entre `texte1[:len(texte1)] = texte1` et `texte2[:len(texte2)] = texte2`.
D'où le résultat voulu.

Le dictionnaire renvoyé, `dist_final`, a pour clés les sommets extraits par la file de priorité utilisée par l'algorithme de Dijkstra (ces clés sont stockées dans le dictionnaire `vue`, alimenté à chaque extraction d'un sommet de la file de priorité). Pour chacun de ses sommets, il dispose de la plus courte distance du sommet considéré au sommet de départ, et donc par ricochet, la distance d'édition entre `texte1[:i]` et `texte2[:j]` (où (i,j) sont les coordonnées du sommet considéré).

Q16 Soit $|S|$ le nombre de sommets du graphe et $|A|$ son nombre d'arêtes.
La complexité de l'algorithme de Dijkstra avec la file de priorité donnée est un $\mathcal{O}((|S|+|A|)\ \log\ |S|)$.
Comme $|A|$ peut être majoré ici par $3|S|$, la complexité est en $\mathcal{O}(|S|\ \log\ |S|)$.
Comme $|S|$ `= len(texte1)`$\times$`len(texte2)`, la complexité de l'algorithme de Dijkstra est ici en $\boxed{\mathcal{O}((\texttt{len(texte1)}\times\texttt{len(texte2)})\ \log(\texttt{len(texte1)}\times\texttt{len(texte2)}))}$.

L'algorithme de Dijkstra, avec une structure de file de complexité quasiment optimale, comme celle proposée par l'énoncé, dispose d'une complexité plus importante que celle de la programmation dynamique de la partie II pour le calcul de la distance d'édition.

Par contre, dans le cas où l'algorithme de Dijkstra ne parcourt pas toute la matrice (par exemple, dans le cas d'un long préfixe commun, tel que `aaaaa` et `aaaab`), il peut être plus intéressant à utiliser que de la programmation dynamique dont l'algorithme parcourt toute la matrice de Levenshtein.

Q17 On peut utiliser pour l'heuristique `h(texte1, texte2, sommet)` la fonction suivante :

> Soient n_1 `= len(texte1)` et n_2 `= len(texte2)`.
> Soit (i,j) les coordonnées du sommet `sommet`.
> `h(texte1, texte2, sommet)` renvoie $max(n_1-i,\ n_2-j) - min(n_1-i,\ n_2-j)$.

La complexité de la fonction `h` est trivialement en $\mathcal{O}(1)$.

Montrons qu'il s'agit d'une heuristique admissible :
On considère un nouveau graphe dont toutes les diagonales $(i,j) \to (i+1,j+1)$ existent (on a relâché une contrainte de l'énoncé). Un chemin du sommet (i,j) à la sortie sera donc de longueur inférieure à la longueur pondérée d'un plus court chemin du graphe initial.
En effet, le chemin pourra éventuellement emprunter une diagonale, là ou il ne le peut pas dans le graphe initial.

Dans ce nouveau graphe, les longueurs de plus court chemins s'obtiennent en considérant « le bord » le plus proche de (i,j),

- s'il s'agit du bord de droite (i.e. $n_2-j < n_1-i$), la longueur du plus court chemin est $(n_1-i)-(n_2-j)$,
- sinon, il s'agit du bord du bas (i.e. $n_1-i \leqslant n_2-j$), la longueur du plus court chemin est $(n_2-j)-(n_1-i)$,

Justifions sur un exemple le gain de temps calcul en comparant les dictionnaires `dist_final` renvoyés par les deux algorithmes.

On considère `texte1 = ['A', 'B', 'C']` et `texte2 = ['B', 'X']`
avec n_1 `= len(texte1) = 3` et n_2 `= len(texte2) = 2`.

— Algorithme de Dijkstra

- Au départ, la file extrait le sommet $(0,0)$, de distance 0, aux sommets qui seront récupérés en fin par `dist_final`.
  L'algorithme ajoute à la file les voisins de $(0,0)$, à savoir $(0,1)$ et $(1,0)$, avec leur distance mise à jour.
- Puis, la file compare les distances de ses éléments $(0,1)$ et $(1,0)$ pour extraire celui de plus faible distance. Comme ces deux sommets ont la même distance 1, la file extrait le plus petit sommet selon l'ordre lexicographique, càd. le sommet $(0,1)$ et l'ajoute avec sa distance aux sommets qui seront récupérés en fin par `dist_final`.

  Puis, l'algorithme ajoute à la file les voisins de $(0,1)$, à savoir $(0,2)$ et $(1,1)$, avec leur distance mise à jour.
- Puis, la file compare les distances de ses éléments $(1,0)$, $(0,2)$ et $(1,1)$ pour extraire celui de plus faible distance. Il s'agit de $(1,0)$, de distance 1. La file l'ajoute avec sa distance aux sommets qui seront récupérés en fin par `dist_final`.
  Puis, l'algorithme ajoute à la file les voisins de $(1,0)$, à savoir $(2,0)$ et $(2,1)$ (notons que $(1,1)$ s'y trouve déjà), avec leur distance mise à jour.
  La file contient à ce moment là : $((2,1),1),((2,0),2),((1,1),2),((0,2),2)$.

On itère jusqu'au sommet d'arrivée. Alors, `dist_final` est égal à
$\{(0,0):0,(0,1):1,(1,0):1,(2,1):1,(0,2):2,(1,1):2,(2,0):2,(2,2):2,(3,1):2,$
$(1,2):3,(3,0):3,(3,2):3\}$.

— Algorithme A^*

- Au départ, la file extrait le sommet $(0,0)$, de coût heuristique nul, et l'ajoute, avec sa distance de 0, aux sommets qui seront récupérés en fin par `dist_final`.
  L'algorithme ajoute à la file les voisins de $(0,0)$, à savoir $(0,1)$ et $(1,0)$, avec leurs coûts heuristiques mise à jour.
  - Pour $(0,1)$:
    `d + h(texte1, texte2, voisin) = 1 + max(3-0, 2-1) - min(3-0, 2-1)`,
    donc 3.
  - Pour $(1,0)$:
    `d + h(texte1, texte2, voisin) = 1 + max(3-1, 2-0) - min(3-1, 2-0)`,
    donc 1.
- Puis, la file compare les coûts heuristiques de ses éléments $(1,0)$, $(0,1)$ pour extraire celui de plus faible coût. Elle extrait le sommet $(1,0)$, de coût 1, et l'ajoute, avec sa distance de 1, aux sommets qui seront récupérés en fin par `dist_final`.

  Puis, l'algorithme ajoute à la file les voisins de $(1,0)$, à savoir $(1,1)$, $(2,0)$ et $(2,1)$, avec leurs coûts heuristiques.
  - Pour $(1,1)$:
    `d + h(texte1, texte2, voisin) = 2 + max(3-1, 2-1) - min(3-1, 2-1)`,
    donc 3.
  - Pour $(2,0)$:
    `d + h(texte1, texte2, voisin) = 2 + max(3-2, 2-0) - min(3-2, 2-0)`,
    donc 3.

- Pour $(2, 1)$:
  `d + h(texte1, texte2, voisin) = 1 + max(3-2, 2-1) - min(3-2, 2-1)`, donc 1.

- Puis, la file compare les coûts heuristiques de ses éléments, à savoir $(0, 1)$, $(1, 1)$, $(2, 0)$ et $(2, 1)$. Elle extrait le sommet $(2, 1)$, de coût 1, et l'ajoute, avec sa distance de 1, aux sommets qui seront récupérés en fin par `dist_final`.

  Puis, l'algorithme ajoute à la file les voisins de $(2, 1)$, à savoir $(2, 2)$ et $(3, 1)$, avec leurs coûts heuristiques.

  - Pour $(2, 2)$:
    `d + h(texte1, texte2, voisin) = 2 + max(3-2, 2-2) - min(3-2, 2-2)`, donc 3.
  - Pour $(3, 1)$:
    `d + h(texte1, texte2, voisin) = 2 + max(3-3, 2-1) - min(3-3, 2-1)`, donc 3.

- Puis, la file compare les coûts heuristiques de ses éléments, à savoir $(0, 1)$, $(1, 1)$, $(2, 0)$, $(2, 2)$ et $(3, 1)$. Comme tous les sommets ont le même coût 3, la file extrait le plus petit sommet selon l'ordre lexicographique, càd. le sommet $(0, 1)$ et l'ajoute, avec sa distance de 1, aux sommets qui seront récupérés en fin par `dist_final`.

  Puis, l'algorithme ajoute à la file les voisins de $(0, 1)$, à savoir $(0, 2)$ (et pas $(1, 1)$ car il y est déjà), avec son coût heuristique.

  - Pour $(0, 2)$:
    `d + h(texte1, texte2, voisin) = 2 + max(3-0, 2-2) - min(3-0, 2-2)`, donc 5.

- Puis, la file compare les coûts heuristiques de ses éléments, à savoir $(1, 1)$, $(2, 0)$, $(2, 2)$, $(3, 1)$, $(0, 2)$. Comme, en dehors de $(0, 2)$, les sommets ont le même coût 3, la file extrait le plus petit sommet selon l'ordre lexicographique, càd. le sommet $(1, 1)$ et l'ajoute, avec sa distance de 2, aux sommets qui seront récupérés en fin par `dist_final`.

  Puis, l'algorithme ajoute à la file les voisins de $(1, 1)$, à savoir $(1, 2)$ (car $(2, 1)$ y a déjà été inséré), avec sa distance mise à jour.

  - Pour $(1, 2)$:
    `d + h(texte1, texte2, voisin) = 3 + max(3-1, 2-2) - min(3-1, 2-2)`, donc 5.

- Puis, la file compare les coûts heuristiques de ses éléments, à savoir $(2, 0)$, $(2, 2)$, $(3, 1)$, $(0, 2)$, $(1, 2)$. Comme, en dehors de $(0, 2)$ et $(1, 2)$, les sommets ont le même coût 3, la file extrait le plus petit sommet selon l'ordre lexicographique [1], càd. le sommet $(2, 0)$ et l'ajoute, avec sa distance de 2, aux sommets qui seront récupérés en fin par `dist_final`.

  Puis, l'algorithme ajoute à la file les voisins de $(2, 0)$, à savoir $(3, 0)$ (car $(2, 1)$ y a déjà été inséré) avec sa distance mise à jour.

  - Pour $(3, 0)$:
    `d + h(texte1, texte2, voisin) = 3 + max(3-3, 2-0) - min(3-3, 2-0)`, donc 5.

- Puis, la file compare les coûts heuristiques de ses éléments, à savoir $(2, 2)$, $(3, 1)$, $(0, 2)$, $(1, 2)$ et $(3, 0)$. Les sommets $(2, 2)$ et $(3, 1)$ ont le même coût 3, la file extrait

1. avec l'ordre lexicographique donné...qui n'en est pas un !

le plus petit sommet selon l'ordre lexicographique, càd. le sommet $(2,2)$ et l'ajoute, avec sa distance de 2, aux sommets qui seront récupérés en fin par `dist_final`.

Puis, l'algorithme ajoute à la file les voisins de $(2,2)$, à savoir $(3,2)$ avec sa distance mise à jour.

- Pour $(3,2)$:
  `d + h(texte1, texte2, voisin) = 3 + max(3-3, 2-2) - min(3-3, 2-2)`, donc 3.

- Puis, la file compare les coûts heuristiques de ses éléments, à savoir $(3,1)$, $(0,2)$, $(1,2)$, $(3,2)$ et $(3,0)$. Les sommets $(3,1)$ et $(3,2)$ ont le même coût 3, la file extrait le plus petit sommet selon l'ordre lexicographique, càd. le sommet $(3,1)$ et l'ajoute, avec sa distance de 2, aux sommets qui seront récupérés en fin par `dist_final`. Aucun autre élément n'est ajouté à la file car le sommet $(3,2)$ y a déjà été ajouté.
- Puis, la file compare les coûts heuristiques de ses éléments, à savoir $(0,2)$, $(1,2)$, $(3,2)$ et $(3,0)$. Elle extrait le sommet $(3,2)$ et l'ajoute, avec sa distance de 3, aux sommets qui seront récupérés en fin par `dist_final`.

  Aucun autre élément n'est ajouté à la fin car le sommet $(3,2)$ n'a pas de voisin (sommet d'arrivée).

Ainsi, `dist_final` est égal à
$\underline{\{(0,0):0,(1,0):1,(2,1):1,(0,1):1,(1,1):2,(2,0):2,(2,2):2,(3,1):2,(3,2):3\}}$.

On a un gain de temps dans l'algorithme A^*, car <u>seuls 9 sommets ont été extraits, au lieu de 12 sommets</u>.

<u>NB :</u> En cas d'égalité, un heapdict fournit le dernier élément inséré, ce qui fournit un meilleur résultat, avec seulement 5 sommets.
Ainsi, ***`dist_final`*** *est égal à* $\{(0,0):0,(1,0):1,(2,1):1,(2,2):2,(3,2):3\}$.