À rendre mercredi 12 mars 2025

Devoir Maison n°6 : IA et révisions

- Pour toute question, n'hésitez pas à me contacter par mail : nvalade.pci@gmail.com ou par cahier-de-prepa.
- On s'interdit l'emploi du test `if x in L` (avec `L` une liste). En revanche le test `if x in D` (avec `D` dictionnaire) est autorisé.
- **Travail à rendre :**
  - Pour Stessy, Lina, Morgane, Hadrien, Fatma, Manon, Hanna, Lorinda (et pour celles et ceux qui se sentent vraiment en très grande difficulté en informatique) :
    - Travaillez à fond la fiche « les indispensables » . Entrainez-vous à traiter les différentes questions (conseil : en faire 3-4 chaque jour).
    - Exercice n°1 : questions 1, 2, 3, 5, 6 et 7.
    - Exercice n°2 : en entier
    - Exercice n°3 : sauf Q9, Q10 et Q16.
  - Pour tou.te.s : exercices n°1 et n°2.
  - Au choix :
    - exercice n°3 (petites questions d'entraînement/révision),
    - ou exercice n°4 (sujet de concours plus difficile, obligatoire pour : Edouard, Ouadi, Younes, Aurélien, Corentin, Thibault C., Baptiste, Ethan).

Exercice n°1 À consommer avec modération !

La professeure de chimie de PCSI a enseigné qu'il y avait trois classes (notées 0,1 et 2) d'alcools dans le vin. Elle a aussi expliqué comment, à l'aide des différentes caractéristiques chimiques, on pouvait déterminer la classe du vin.

Seulement, Émile, fils de vigneron et donc fin connaisseur des alcools, n'a pas écouté la méthode pour déterminer la classe du vin[1]. Ainsi, il a une liste de 178 vins avec chacune les différentes caractéristiques chimiques. Mais il a seulement la classe des vins pour exactement la moitié : ceux d'indice compris 0 et 88 (en effet, Émile s'est endormi à la moitié du cours).

Il va falloir aider Émile à reconstituer, du mieux possible, les classes manquantes (de 89 à 177). Pour cela, l'algorithme des k-plus proche voisins semble tout approprié. En effet, si un vin donné a parmi les 3 plus proches vins, 2 vins de classe 1 et un vin de classe 0 , on pourra en déduire que ce vin est probablement de classe 1 .

On a chargé les données de Émile dans Python grâce aux variables `Li` et `T` suivantes :

- `Li` est la liste des caractéristiques des vins (de longueur 178) ;
- `Li[i]` correspond aux caractéristiques du vin $n°i$ de la liste `Li` ;
- `Li[i]` est lui-même un tuple de flottants correspondants aux différentes caractéristiques (le degré d'alcool, la quantité de magnésium,...) ;
- `T` est la liste des classes connues par Émile des vins (de longueur 89 donc) telle que pour tout $i \in [\![0, 88]\!]$, `T[i]` est la classe (notée 0,1 ou 2) du vin n°i de caractéristiques `Li[i]`.
- `Caracteristiques` est la liste des chaines de caractères détaillant à quoi correspond chaque élément d'un tuple de `Li` (un vin).

Par exemple, comme `Li[0]` la liste des caractéristiques du vin numéro 0 :

- comme `Caractéristiques[0]` est égal à `'alcohol'` et que `Li[0][0]=14.23`, on peut dire que le degré d'alcool du vin numéro 0 est de 14.23.

1. Une partie de l'histoire est vraie...

- comme `Caractéristiques[4]` est égal à `'magnesium'` et que `Li[0][4]=127`, on peut dire que la quantité de magnésium du vin numéro 0 est de 127 (dans une certaine unité que Émile n'a pas notée).
- Par contre, Émile a bien noté que la valeur de `T[0]` (comme pour tous les vins numérotés jusqu'à 88). Celle-ci vaut 1 , donc le vin numéro 0 a pour classe 1 .

Q1. Écrire une fonction `Dist(M:list, N:list)->float`, où `M` et `N` sont deux listes de même longueur, renvoie la distance entre ces deux listes (par exemple les caractéristiques de deux vins) à savoir :

$$\sqrt{\sum_{k=0}^{\text{len}(M)-1} (M[k]-N[k])^2}$$

Q2. Écrire une fonction `DicoOccurrences(L:list)->dict` qui, à une liste `L`, renvoie le dictionnaires du nombre d'occurrences, c'est-à-dire que les clés de ce dictionnaire sont les éléments de la liste et la valeur associée à une clé est le nombre d'occurrences de cet élément dans `L`.

Q3. En déduire une fonction `Majoritaire(L:list)->int` qui renvoie un élément dont le nombre d'occurrences est majoritaire.

Q4. Écrire la fonction `PlusProchesVoisins (Li:list,T:list,i:int,k:int)->int` qui renvoie la classe majoritaire parmi les k plus proches voisins de `Li[i]` ($89 \leqslant i < 178$).

Idée :

- Constituer le dictionnaire des distances : les clés sont les numéros j des alcools dont la classe est connue ($0 \leqslant j \leqslant 88$), et les valeurs sont les distances qui les séparent de l'alcool `i` de classe inconnue.
- Utiliser la fonction `sorted()` de Python qui accepte en entrée des listes mais aussi des dictionnaires et renvoie, dans le cas d'un dictionnaire, la liste des clés de celui-ci triée par ordre de valeur associée croissante.
- En déduire la classe majoritaire parmi les `k` plus proches voisins de `i`.

L'enseignante de chimie a enfin pitié du pauvre Émile et décide de lui donner les classes manquantes.
La liste `T` comporte donc maintenant 178 éléments. Grâce à cela il peut en déduire les matrices de confusion pour différentes valeurs de k, notées M_k :

$$M_3 = \begin{pmatrix} 24 & 2 & 3 \\ 2 & 24 & 10 \\ 0 & 9 & 14 \end{pmatrix} \quad M_4 = \begin{pmatrix} 25 & 2 & 2 \\ 2 & 23 & 11 \\ 1 & 7 & 15 \end{pmatrix} \quad M_5 = \begin{pmatrix} 28 & 1 & 0 \\ 2 & 26 & 8 \\ 2 & 9 & 12 \end{pmatrix} \quad M_6 = \begin{pmatrix} 27 & 0 & 2 \\ 2 & 23 & 11 \\ 4 & 10 & 9 \end{pmatrix} \quad M_7 = \begin{pmatrix} 28 & 1 & 0 \\ 2 & 27 & 7 \\ 4 & 13 & 6 \end{pmatrix}$$

Q5. Pourquoi les matrices de confusion sont des matrices 3×3 ?

Q6. Quelle est la valeur de k la plus adaptée ? Justifier.

Q7. Pour cette valeur de k, quel pourcentage de réussite avait Émile ?

Q8. Écrire une fonction `MatriceDeConfusion(Li:list, T:list,k:int)->list[list]` qui renvoie la matrice de confusion associée à l'entier k donné en argument.

Exercice n°2 Base de données

On considère la base de données suivantes :

- `Region (id_region, Nom_Region, Nb_Cepages)`, `id_region` est la clé primaire, `Nom_Region` est le nom de la région, et `Nb_Cepages` est le nombre de cépages dans chaque région.
- `Appellation (id_app, Nom_Appellation, id_region, Decrets, Nb_Cepages)`, avec `id_app` est la clé primaire, `Nom_Appellation` est le nom de l'appellation, `id_region` est la clé étrangère en relation avec la clé primaire de `Region`, `Decrets` est le numéro du décret attribuant cette appellation, `Nb_Cepages` est le nombre de cépages de chaque appellation.
- `Cepage (id_cepage, Couleur, Nom, Nb_Appellation, Superficie)`, avec `id_cepage` est la clé primaire, `Nb_Appellation` est le nombres d'appellation de chaque cépage, `Nb_Appellation` est le nombre d'appellations de chaque cépage
- `Encepagement (id_app, id_cepage)`, ces attributs sont en relation avec les clés primaires respectivement de `Appellation` et `Cepage`.

Q1. Écrire la requête en SQL qui permet d'obtenir la liste des noms de cépage de couleur "rouge".

Q2. Écrire la requête en SQL qui permet d'obtenir le nombre moyen de cépages par région.

Q3. Écrire la requête en SQL qui permet d'obtenir la liste des noms d'appellations pour le cépage "Chardonnay" de couleur "blanc".

Q4. Écrire la requête en SQL qui permet d'obtenir la liste des surfaces totales pour chaque couleur de cépage.

Q5. Écrire la requête en SQL qui permet d'obtenir la liste des couleurs ayant au moins 5 cépages.

Exercice n°3 Révisons !

Partie I Autour de la moyenne

Q1. Soit `L` une liste de listes de flottants, écrire une fonction `Moyenne(L:list)->float` qui renvoie la moyenne des éléments de cette liste.

Q2. Écrire une fonction `DepasseMoyenne(L:list)->list` qui renvoie la liste des éléments de `L` strictement plus grand que la moyenne

Attention : la fonction `Moyenne(L)` devra être appelée une et une seule fois.

Q3. Quelle est la complexité de ces deux fonctions ?

Partie II Autour des graphes

Q4. Expliquer pourquoi la recherche d'un élément `x` dans un dictionnaire `D` via la commande `if x in D` a une complexité en $\mathcal{O}(1)$. On emploiera de façon adéquate les mots «tables de hachage» et «fonction de hachage » en expliquant leur utilité et le principe.

Considérons le graphe définie le dictionnaire d'adjacence :

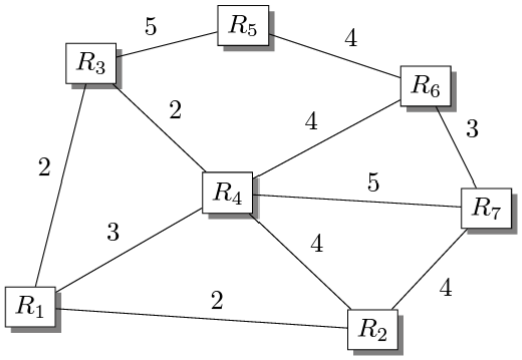
```
Dico={"A" : ["B","C","D"], "B" : ["A","C","E"], "C" : [], "D" : ["B","C"], "E" : ["C","D"] }
```

Q5. Faites un dessin du graphe.

Q6. Donner la liste des points qui seront visités si on fait un parcours en largeur en partant de `A`.

Q7. Donner la liste des points qui seront visités si on fait un parcours en profondeur partant de `B`.

On considère un graphe qui représente une partie d'un réseau constitué de sept appareils. On cherche la route la plus courte entre l'appareil R1 et l'appareil R6.



Ce réseau est défini par le dictionnaire `reseau` et l'heuristique est définie par le dictionnaire `heur` :

```
reseau={"R1" : [ ("R2",2) , ("R3",2) , ("R4",3) ] ,
        "R2" : [ ("R1",1) , ("R4",4) , ("R7",4)],
        "R3" : [ ("R1",2),("R4",2),("R5",5)],
        "R4" : [("R1",3),("R2",4),("R3",2),("R6",4),("R7",5) ],
        "R5" : [("R3",5),("R6",4)],
        "R6" : [("R4",4),("R5",4),('R7',3)],
        "R7" : [("R2",4),('R4',5),('R6',3)]   }
heur = { "R1":6 , "R2":6 , "R3":6 , "R4":3 , "R5":3 ,  "R6":0 , "R7":3 }
```

Q8. On cherche les plus courtes distances entre chaque sommet et R1. Appliquer l'algorithme de Dijkstra « à la main » en construisant un tableau des distances.

Q9. On cherche la plus courte distance entre R1 et R6. Appliquer l'algorithme A^* « à la main » en construisant un tableau des distances. L'**heuristique** choisie est le nombre d'arêtes reliant les sommets multiplié par 3, où 3 est la longueur moyenne supposée d'une arête.

Q10. Comparer avec les étapes d'un programme utilisant l'algorithme A^*.

Partie III Autour des matrices (tableaux à deux dimensions)

Une matrice est modélisée par une liste de listes.

Q11. Écrire une fonction `MatriceNulle(n:int,p:int)->list[list]` qui renvoie la matrice nulle à n lignes et p colonnes.

Q12. Écrire une fonction `MatriceDamier(n:int,p:int)->list[list]` qui renvoie la matrice à n lignes et p colonnes dont les coefficients valent 1 si la somme de leur indice de ligne et celui de colonne est paire et 0 sinon.

Ainsi `MatriceDamier(2,3)` renvoie `[[1,0,1],[0,1,0]]`.

Q13. Écrire une fonction `MaxCoeffMatrice(M:list[list])->float` qui, à une matrice `M`, renvoie le plus grand coefficient de `M`.

Q14. Écrire une fonction `Transposee(M:list[list])->list[list]` qui renvoie la transposée de la matrice `M`.

Q15. Écrire une fonction `Produit(A:list[list],B:list[list])->list[list]` qui, à deux matrices `A` et `B`, renvoie la matrice AB. Utiliser un `assert` pour vérifier que les tailles des matrices sont bien compatibles.

On rappelle que le coefficient de AB à la i-ième ligne et la j-ième colonne est : $\sum_{k=0}^{p-1} A[i][k] \times B[k][j]$, où p est le nombre de colonnes de A (et le nombre de lignes de B). L'indice commence à 0 en Python.

Q16. Écrire une fonction `Extraire(M:list[list],i:int,j:int)->list[list]` qui renvoie la matrice issue de `M` sans sa i-ième ligne ni sa j-ième colonne.

Exercice n°4 Grands textes

Dans ce sujet, on s'intéresse à des textes de grande taille auxquels plusieurs auteurs apportent des modifications au cours du temps. Ces textes peuvent par exemple être des programmes informatiques développés par de multiples auteurs. Il est important de pouvoir efficacement gérer les différentes versions de ces programmes au cours de leur développement et limiter le stockage et la transmission d'informations redondantes. Nous allons pour cela nous intéresser à une notion de différentiels entre textes.

Complexité.

La complexité, ou le temps d'exécution, d'une fonction P est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc...) nécessaires à l'exécution de P dans le cas le pire. Lorsque la complexité dépend d'un ou plusieurs paramètres $\kappa_1, \cdots, \kappa_r$, on dit que P a une complexité en $\mathcal{O}\left(f\left(\kappa_1, \cdots, \kappa_r\right)\right)$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $\kappa_1, \cdots, \kappa_r$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $\kappa_1, \cdots, \kappa_r$, la complexité est au plus $C \cdot f\left(\kappa_1, \cdots, \kappa_r\right)$.

Lorsqu'il est demandé de donner la complexité d'un programme, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Rappels concernant le langage Python.

Ce sujet utilise les types `Python` listes et dictionnaires, mais seules les opérations mentionnées ci-dessous sont autorisées dans vos réponses.

Si `l`, `l1`, `l2` désignent des listes en `Python` :

- `len(l)` renvoie la longueur de la liste `l`, c'est-à-dire le nombre d'éléments qu'elle contient. Complexité en $\mathcal{O}(1)$.
- `l1 == l2` teste l'égalité des listes `l1` et `l2`. Complexité en $\mathcal{O}(n)$ avec n le minimum de `len(l1)` et `len(l2)`.
- `l[i]` désigne le i-ème élément de la liste `l`, où l'indice `i` est compris entre `0` et `len(l)-1`. Complexité en $\mathcal{O}(1)$.
- `l[i :j]` construit la sous-liste `[l[i], ..., l[j-1]]`. Complexité en $\mathcal{O}(j-i)$. L'usage des variantes `l[i :]` à la place de `l[i :len(l)]`, et de `l[ :j]` à la place de `l[0 :j]` est aussi autorisé.
- `l.append(e)` modifie la liste `l` en lui ajoutant l'élément `e` en dernière position. Complexité en $\mathcal{O}(1)$.
- `l.pop()` renvoie le dernier élément de la liste `l` (supposée non vide) et supprime l'occurrence de cet élément en dernière position dans la liste. Complexité en $\mathcal{O}(1)$.

On pourra aussi utiliser la fonction `range` pour réaliser des itérations.

Si `d` est un dictionnaire `Python` :

- `{key_1 : v_1, ..., key_n : v_n }` crée un nouveau dictionnaire en associant chaque valeur `v_i` à une clé `key_i`. Complexité en $\mathcal{O}(n)$.
- `d[key]` renvoie la valeur associée à la clé `key` dans `d` et lève une erreur si la clé `key` n'est pas présente. Complexité en $\mathcal{O}(1)$.
- `d[key] = v` modifie `d` pour associer la valeur `v` à la clé `key`, même si la clé `key` n'est pas présente dans `d` initialement. Complexité en $\mathcal{O}(1)$.
- `key in d` teste si la clé `key` est présente dans `d`. Complexité en $\mathcal{O}(1)$.

Sauf mention contraire, les fonctions à écrire ne doivent pas modifier leurs entrées.

La structure de données texte. Dans ce sujet, on appelle texte une liste de caractères.

Par exemple, `['b', 'i', 'n', 'g', 'o']` est un texte de longueur 5.

Partie I Différentiels par positions fixes

Dans cette partie, nous traitons le problème avec une hypothèse simplificatrice : les textes comparés ont toujours la même taille.

Q1. Sans utiliser le test `==` sur les listes, écrire une fonction `textes_égaux(texte1, texte2)` qui teste si deux textes sont égaux. Donner la complexité de cette fonction.

Exemples :

```
>>>textes_égaux(['v', 'i', 's', 'a'], ['v', 'a', 'i', 's'])}
False
>>>textes_égaux(['v', 'i', 's', 'a'], ['v', 'i', 's', 'a'])}
True
```

Dans la suite de ce sujet, on pourra utiliser `==` sur les listes plutôt que cette fonction.
Si deux textes ne sont pas égaux mais ont la même longueur n, on souhaite compter le nombre de positions qui diffèrent, c'est à dire déterminer combien il existe de positions i ($0 \leq i < n$) telles que les caractères en position i sont différents dans les deux textes.

Q2. Écrire une fonction `distance(texte1, texte2)` qui calcule cette quantité. On supposera que les deux textes ont le même nombre de caractères. Donner la complexité de cette fonction.

Exemples

```
>>> distance(['v', 'i', 's', 'a'], ['v', 'a', 'i', 's'])
3
>>> distance(['a', 'v', 'i', 's'], ['v', 'i', 's', 'a'])
4
```

Q3. En vous aidant d'un dictionnaire dont les clés sont des caractères, écrire une fonction `aucun_caractère_commun(texte1, texte2)` qui renvoie `True` si et seulement si l'ensemble des caractères qui apparaissent dans `texte1` est disjoint de l'ensemble des caractères qui apparaissent dans `texte2`. Les deux textes peuvent avoir ici des longueurs différentes.

Cette fonction devra avoir une complexité $\mathcal{O}(\texttt{len(texte1)} + \texttt{len(texte2)})$.

Exemples

```
>>> aucun_caractère_commun(['a', 'v', 'i', 's'], ['v', 'i', 's', 'a'])
False
>>> aucun_caractère_commun(['a', 'v', 'i', 's'], ['u', 'r', 'n', 'e'])
True
```

Nous introduisons maintenant une structure de données spécifique pour représenter un différentiel par positions fixes entre deux textes.

La FIGURE 1 présente un exemple de couple de textes $(\texttt{texte}_1, \texttt{texte}_2)$ qui diffèrent sur 4 tranches (représentées par des zones grisées sur la FIGURE 1). En dehors des tranches, les textes sont égaux.

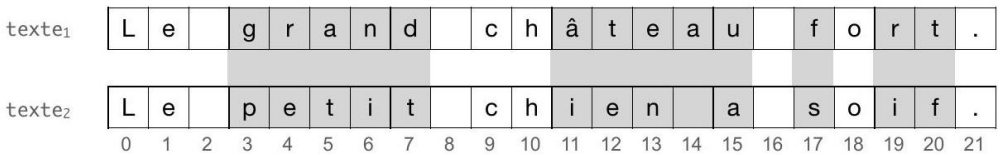


FIGURE 1 - Exemple de couple $(\texttt{texte}_1, \texttt{texte}_2)$ dont on veut calculer le différentiel (sur des positions fixes).

La structure de données tranche. Une tranche est un dictionnaire avec trois clés `'début'`, `'avant'` et `'après'`. La valeur associée à la clé `'début'` est le premier indice de la tranche, les textes associés aux clés `'avant'` et `'après'` représentent les textes (**de même longueur**) de la tranche avant et après modification. Dans la suite de cette partie, on s'appuiera sur les fonctions suivantes pour manipuler cette structure.

```
def tranche(arg_début, arg_avant, arg_après):
    return {'début': arg_début, 'avant': arg_avant, 'après': arg_après}
def début(tr):
    return tr['début']
def après(tr):
    return tr['après']
def avant(tr):
    return tr['avant']
def fin(tr):
    return début(tr) + len(après(tr))
```

Nous ne fournissons pas de fonction pour modifier une tranche car nous souhaitons traiter cette structure de données comme une structure immuable[1].
On peut représenter le différentiel de la FIGURE 1, par la liste suivante :

```
[
tranche(3, ['g', 'r', 'a', 'n', 'd'], ['p', 'e', 't', 'i', 't']),
tranche(11, ['â', 't', 'e', 'a', 'u'], ['i', 'e', 'n',' ','a']),
tranche(17, ['f'],['s']),
tranche(19, [' r ', 't'], ['i', 'f'])
]
```

La structure de données différentiel. Un différentiel est une liste (potentiellement vide) de tranches `[tr`$_1$`, ..., tr`$_k$`]` représentant des modifications touchant des zones distinctes d'un texte, telle que

- `début(tr`$_1$`) < fin(tr`$_1$`) < ...< début(tr`$_k$`) < fin(tr`$_k$`)`
- pour tout $j \in [1, k]$, pour tout $i \in [0,$ `len(avant(tr`$_j$`))-1]`, `avant(tr`$_j$`)[i]` $\neq$ `après(tr`$_j$`)[i]`

Existence et unicité d'un différentiel par positions fixes. Pour deux textes `texte`$_1$ et `texte`$_2$ de même longueur n, il existe un unique différentiel `[tr`$_1$`, ..., tr`$_k$`]` tel que :

- si $k > 0$, alors $0 \leq$ `début(tr`$_1$`)` et `fin(tr`$_k$`)`$\leq n$
- pour tout $j \in [1, k]$, `texte`$_1$`[début(tr`$_j$`) : fin(tr`$_j$`)] = avant(tr`$_j$`)`
- pour tout $j \in [1, k]$, `texte`$_2$`[début(tr`$_j$`) : fin(tr`$_j$`)] = après(tr`$_j$`)`
- pour tout $i \in [0, n-1]$, si $i \notin \bigcup_{1 \leq j \leq k}$ `[debut(tr`$_j$`), fin(tr`$_j$`)-1]`, alors `texte`$_1$`[i] = texte`$_2$`[i]`

Cet unique différentiel est appelé le différentiel de `texte`$_2$ vis-à-vis de `texte`$_1$.
Toutes les propriétés précédentes sur les différentiels assurent les propriétés intuitives suivantes :

- les tranches sont présentées par indices de début croissants, sans se chevaucher, ni se toucher ;
- chaque tranche `tr` couvre un intervalle de positions `[début(tr),fin(tr) - 1]` sur lequel `texte`$_1$ et `texte`$_2$ diffèrent à chaque position, et dont les sous-textes sur ces intervalles correspondent à `avant(tr)` pour `texte`$_1$ et `après(tr)` pour `texte`$_2$.

Q4. Écrire une fonction `différentiel(texte1, texte2)` qui calcule le différentiel du texte `texte2` vis-à-vis du texte `texte1`, supposés de même longueur. La complexité attendue est $\mathcal{O}$`(len(texte1))`. Justifier cette complexité.

Q5. Écrire une fonction `applique(texte1, diff)` qui, étant donné un texte `texte1` et un différentiel `diff`, renvoie un texte `texte2` tel que `diff` soit le différentiel de `texte2` vis-à-vis de `texte1`. On supposera que le différentiel `diff` contient des tranches cohérentes avec la taille et le contenu du texte `texte1`. Donner et justifier la complexité.

Pour reconstruire l'ancienne version d'un texte à partir d'un différentiel, nous allons nous appuyer sur la notion de différentiel inversé.

Q6. Écrire une fonction `inverse(diff)` telle que pour tous textes `texte1`, `texte2` de même longueur, si `diff` désigne `différentiel(texte1, texte2)`, alors `applique(texte2, inverse(diff))` = `texte1` et `inverse(inverse(diff))` = `diff`. Donner sa complexité.

La structure de données *texte versionné*. Nous représentons un texte versionné par un dictionnaire contenant la version courante du texte, comme valeur associée à la clé `'courant'`, et l'historique des différentiels qui ont mené jusqu'à cette version dans une pile[2] de différentiels associée à la clé `historique`. Dans la suite de cette partie, on s'appuiera sur les fonctions suivantes pour manipuler cette structure.

```
def versionne(texte):
    return {'courant' : texte, 'historique' : [] }
def courant(texte_versionné):
    return texte_versionné['courant']
def remplace_courant(texte_versionné, texte):
    texte_versionné['courant'] = texte
def historique(texte_versionné):
    return texte_versionné['historique']
```

Contrairement à la structure immuable de tranche, nous nous autorisons cette fois à modifier la structure de texte versionné, en particulier la pile qu'elle contient via les opérations `historique(texte_versionné).append(diff)` et `historique(texte_versionné).pop()`.

Q7. Écrire les fonctions `modifie(texte_versionné, texte)` et `annule(texte_versionné)` qui assurent les deux opérations de base attendues sur un texte versionné `texte_versionné`. La fonction `modifie(texte_versionné, texte)` modifie `texte_versionné` pour lui ajouter une nouvelle version correspondant au texte `texte`, en supposant qu'il a la même longueur n que le texte courant. La taille de l'historique augmente alors de 1. La fonction ne renvoie rien. La fonction `annule(texte_versionné)` modifie `texte_versionné` en annulant l'effet de la dernière modification effectuée et renvoie la nouvelle valeur courante du texte. La taille de l'historique diminue alors de 1. On suppose que la pile des différentiels n'est pas vide lors de cet appel. Donnez les complexités de ces deux fonctions.

Exemples

```
>>> texte_versionné = versionne(['a', 'v', 'i', 's'])
>>> modifie(texte_versionné, ['v', 'i', 's', 'a'])
>>> modifie(texte_versionné, ['v', 'i', 't', 'a'])
>>> modifie(texte_versionné, ['l', 'i', 's', 'a'])
>>> assert courant(texte_versionné) == ['l', 'i', 's', 'a']
>>> assert historique(texte_versionné) == [
        différentiel(['a', 'v', 'i', 's'], ['v', 'i', 's', 'a']),
        différentiel(['v', 'i', 's', 'a'], ['v', 'i', 't', 'a']),
        différentiel(['v', 'i', 't', 'a'], ['l', 'i', 's', 'a'])
]
>>> annule(texte_versionné)
['v', 'i', 't', 'a']
>>> annule(texte_versionné)
['v', 'i', 's', 'a']
>>> annule(texte_versionné)
['a', 'v', 'i', 's']
```

Partie II Différentiels sur des positions variables

Dans cette partie, nous nous intéressons à des différentiels de textes dont les longueurs ne sont plus forcément égales. Nous adaptons pour cela la définition de tranche et de différentiel. La FIGURE 2 présente un exemple de couple (`texte`$_1$, `texte`$_2$) dont on va représenter le différentiel par une liste de tranches (représentées par des zones grisées sur la figure). Cette fois, les tranches désignent des portions de textes qui ne sont pas nécessairement de la même longueur, ni alignées.

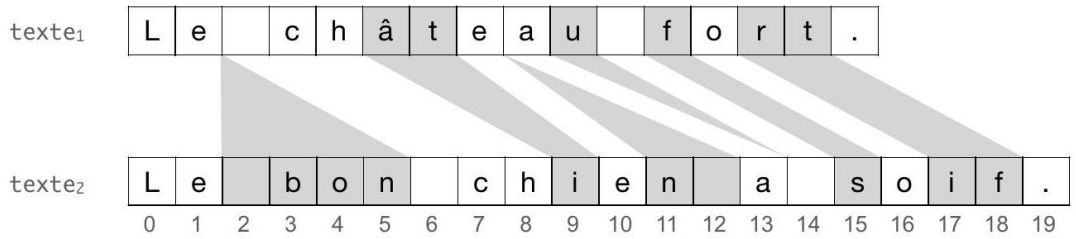


FIGURE 2 - Exemple de couple (texte$_1$ texte$_2$) dont on veut calculer le différentiel sur des positions variables.

Nouvelle structure de données tranche. Un différentiel d'un texte `texte`$_2$ vis-à-vis d'un texte `texte`$_1$ est toujours une liste de tranches mais chaque tranche comporte maintenant 4 clés :

- la clé `'début_avant'` représente la position i d'un sous-texte avant qui a été supprimé de `texte`$_1$;
- la clé `'avant'` est associée au texte avant ;
- la clé `'début_après'` représente la position dans `texte`$_2$ d'un sous-texte après, qui a été ajouté à la place du sous-texte avant en position i dans `texte`$_1$;
- la clé `'après'` est associée au texte après.

Dans la suite de cette partie, on s'appuiera sur les fonctions suivantes pour manipuler cette structure.

```
def tranche(arg_début_avant, arg_avant, arg_début_après, arg_après):
    return {'début_avant': arg_début_avant,
            'avant': arg_avant,
            'début_après': arg_début_après,
            'après': arg_après}
def début_avant(tr):
    return tr['début_avant']
def début_après(tr):
    return tr['début_après']
def après(tr):
    return tr['après']
def avant(tr):
    return tr['avant']
def fin_avant(tr):
    return début_avant(tr) + len(avant(tr))
def fin_après(tr):
    return début_après(tr) + len(après(tr))
```

Nouvelle structure de données différentiel. Un différentiel est une liste (potentiellement vide) de tranches [`tr`$_1$, ..., `tr`$_k$] telle que

- `début_avant(tr`$_1$`)` $\leq$ `fin_avant(tr`$_1$`)` $< \ldots <$ `début_avant(tr`$_k$`)` $\leq$ `fin_avant(tr`$_k$`)`
- `début_après(tr`$_1$`)` $\leq$ `fin_après(tr`$_1$`)` $< \ldots <$ `début_après(tr`$_k$`)` $\leq$ `fin_après(tr`$_k$`)`
- pour tout $j \in [1, k]$, `aucun_caractère_commun(avant(tr`$_j$`),après(tr`$_j$`))` = `True`
- pour tout $j \in [1, k]$, `len(avant(tr`$_j$`))` > 0 ou `len(après(tr`$_j$`))` > 0

Notion de différentiel valide vis-à-vis de deux textes. Pour deux textes `texte`$_1$ et `texte`$_2$ de même longueur n, un différentiel valide de `texte`$_2$ vis-à-vis de `texte`$_1$ est une liste `diff` = [`tr`$_1$, ..., `tr`$_k$] de tranches telle que :

- si $k > 0$, alors $0 \leq$ `début_avant(tr`$_1$`)` et `fin_avant(tr`$_k$`)` $\leq$ `len(texte`$_1$`)`
- si $k > 0$, alors $0 \leq$ `début_après(tr`$_1$`)` et `fin_après(tr`$_k$`)` $\leq$ `len(texte`$_2$`)`
- pour tout $j \in [1, k]$, `texte`$_1$`[début_avant(tr`$_j$`) : fin_avant(tr`$_j$`)]` $=$ `avant(tr`$_j$`)`
- pour tout $j \in [1, k]$, `texte`$_2$`[début_après(tr`$_j$`) : fin_après(tr`$_j$`)]` $=$ `après(tr`$_j$`)`
- pour tout $j \in [1, k-1]$, les sous-textes `texte`$_1$`[fin_avant(tr`$_j$`) : début_avant(tr`$_{j+1}$`)]` et `texte`$_2$`[fin_après(tr`$_j$`) : début_après(tr`$_{j+1}$`)]` sont égaux
- si $k = 0$ alors les textes `texte`$_1$ et `texte`$_2$ sont égaux
- si $k > 0$ alors `texte`$_1$`[0 : début_avant(tr`$_1$`)]` $=$ `texte`$_2$`[0 : début_après(tr`$_1$`)]` et `texte`$_1$`[fin_avant(tr`$_k$`) : len(texte`$_1$`)]` $=$ `texte`$_2$`[fin_après(tr`$_k$`) : len(texte`$_2$`))]`

On peut représenter le différentiel de la FIGURE 2, par la liste suivante :

```
[
tranche( 2, [], 2, [' ', 'b', 'o', 'n']),
tranche( 5, ['â', 't'], 9, ['i']),
tranche( 8, [], 11, ['n',' ']),
tranche( 9, ['u'], 14, []),
tranche(11, ['f'], 15, ['s']),
tranche(13, ['r', 't'], 17, ['i', 'f'])
]
```

On admet que, comme dans la partie précédente, on peut écrire des fonctions `applique` et `inverse` satisfaisant les mêmes propriétés que précédemment sur cette nouvelle notion de différentiel. On définit le poids d'un différentiel comme la somme des longueurs des sous-textes `avant(tr)` et `après(tr)` pour toutes les tranches `tr` qui le composent.

Q8. Écrire une fonction `poids(diff)` qui calcule le poids d'un différentiel `diff`. Donner sa complexité.

Exemple

```
>>> poids([tranche(0, ['b'], 0, ['t', 'r', 'o', 't', 't']),
           tranche(2, ['c', 'y', 'c', 'l'], 6, ['n'])])
11
```

On s'intéresse à la distance d'édition[2] entre deux textes. Dans ce sujet, on définit cette distance comme le nombre minimal de suppressions et d'insertions de caractères pour passer d'un texte à un autre. On peut facilement se convaincre que cette distance coïncide avec le poids minimal possible pour un différentiel entre les deux textes.

Nous allons calculer cette distance par programmation dynamique. Pour deux textes `texte`$_1$ et `texte`$_2$ fixés, et pour $0 \leq i \leq$ `len(texte`$_1$`)` et $0 \leq j \leq$ `len(texte`$_2$`)`, on note $M[i][j]$ la distance d'édition pour passer de `texte`$_1$`[0 : `i`]` à `texte`$_2$`[0 : `j`]`. La matrice[4] M est appelée matrice de distance d'édition entre `texte`$_1$ et `texte`$_2$.

La FIGURE 3 présente la matrice M pour `texte`$_1$`=['A', 'B', 'C', 'D', 'C', 'E', 'F']` et `texte`$_2$`=['U', 'A', 'B', 'C', 'C', 'X', 'Y', 'Z']`.

Q9. Donnez une équation de récurrence qui exprime $M[i+1][j+1]$ en fonction de $M[i][j]$, $M[i][j+1]$, $M[i+1][j]$, `texte`$_1$`[i]` et `texte`$_2$`[j]`, pour $0 \leq i <$ `ln(texte`$_1$`)` et $0 \leq j <$ `ln(texte`$_2$`)`. Justifier brièvement la validité de cette équation, sans rédiger une preuve complète.

Q10. Écrire une fonction `levenshtein(texte1, texte2)` de complexité polynomiale qui renvoie la matrice M. Préciser sa complexité.

2. Cette distance est communément appelée distance de Levenshtein.
4. Dans ce sujet, nous représenterons ces matrices par des listes de listes d'entiers.

On utilisera l'instruction `M = [[ 0 for j in range(m)] for i in range(n) ]` pour initialiser une matrice `M` de `n` lignes et `m` colonnes avec des zéros.

		U	A	B	C	C	X	Y	Z
	0	1	2	3	4	5	6	7	8
A	1	2	1	2	3	4	5	6	7
B	2	3	2	1	2	3	4	5	6
C	3	4	3	2	1	2	3	4	5
D	4	5	4	3	2	3	4	5	6
C	5	6	5	4	3	2	3	4	5
E	6	7	6	5	4	3	4	5	6
F	7	8	7	6	5	4	5	6	7

FIGURE 3 - Exemple de matrice de distance d'édition

Q11. Écrire une fonction `différentiel(texte1, texte2, M)` qui calcule un différentiel du texte `texte2` vis-à-vis du texte `texte1`, en s'aidant de la matrice de distance `M` donnée par `levenshtein(texte1, texte2)`. Le différentiel renvoyé doit être de poids minimal.

La fonction devra avoir une complexité $\mathcal{O}(\texttt{len(texte1)} + \texttt{len(texte2)})$. Justifier cette complexité et expliquer brièvement pourquoi le différentiel calculé satisfait les propriétés attendues par un différentiel. On pourra s'aider de la FIGURE 3 pour comprendre quel parcours suivre dans la matrice `M`.

Si on se place dans un scénario de travail collaboratif où deux auteurs différents modifient en parallèle le même texte `texte`, il est nécessaire de pouvoir fusionner leur travail. Nous notons $\texttt{texte}_1$ le nouveau texte obtenu après le travail du premier auteur sur `texte` et `diff1` le différentiel correspondant. De même, nous notons $\texttt{texte}_2$ le texte obtenu après le travail du deuxième auteur sur le même texte `texte`, et `diff2` le différentiel correspondant.

Exemple

```
>>> texte =
    ['l', 'e', ' ', 'c', 'h', 'a', 't', ' ', 'a', ' ', 's', 'o', 'i', 'f'
   ]
>>> texte1 =
        ['l', 'e', ' ', 'c', 'h', 'a', 't', ' ', 'a', ' ', 't', 'r', 'è',
    's', ' ', 's', 'o', 'i', 'f']
>>> texte2 =
        ['l', 'e', ' ', 'c', 'h', 'i', 'e', 'n', ' ', 'a', ' ', 's', 'o',
    'i', 'f']
>>> diff1 = différentiel(texte, texte1, levenshtein(texte, texte1))
>>> assert diff1 == [tranche(9, [], 9, [' ', 't', 'r', 'è', 's'])]
>>> diff2 = différentiel(texte, texte2, levenshtein(texte, texte2))
>>> assert diff2 == [tranche(5, ['a', 't'], 5, ['i', 'e', 'n'])]
```

Pour fusionner le travail des deux auteurs, on apporte des modifications à `diff2` de façon à ce que le texte final, qui inclut les modifications des deux auteurs, soit exprimable comme l'application du différentiel `diff1`, puis de la nouvelle version de `diff2` sur le texte initial. Dans l'exemple précèdent, le texte final attendu est : `['l', 'e', ' ', 'c', 'h', 'i', 'e', 'n',' ', 'a', ' ', 't', 'r', 'è', 's', ' ', 's', 'o', 'i', 'f']`.

Nous allons être prudents en nous assurant au préalable que les modifications apportées ne concernent pas les même zones du texte initial.

Q12. Écrire une fonction `conflit(diff1, diff2)` qui prend en argument deux différentiels `diff1` et `diff2` et renvoie `True` si et seulement s'il existe une tranche $\texttt{tr}_1$ dans `diff1` et une tranche $\texttt{tr}_2$ dans `diff2` telles que

$$[\texttt{début\_avant}(\texttt{tr}_1), \texttt{fin\_avant}(\texttt{tr}_1)] \cap [\texttt{début\_avant}(\texttt{tr}_2), \texttt{fin\_avant}(\texttt{tr}_2)] \neq \emptyset$$

Cette fonction devra avoir une complexité $\mathcal{O}(\texttt{len(diff1)} + \texttt{len(diff2)})$ que l'on justifiera.

Q13. Écrire une fonction `fusionne(diff1, diff2)` qui renvoie un nouveau différentiel représentant la mise à jour de `diff2`. Il est attendu que `poids(fusionne(diff1, diff2))` = `poids(diff2)`. On suppose que les deux différentiels `diff1` et `diff2` ne sont pas en conflit. Cette fonction devra avoir une complexité $\mathcal{O}(\texttt{len(diff1)} + \texttt{len(diff2)})$.

Exemple

```
>>> assert not conflit(diff1, diff2)
>>> print(applique(applique(texte, diff1), fusionne(diff1, diff2)))
['l', 'e', ' ', 'c', 'h', 'i', 'e', 'n',' ', 'a',' ', 't', 'r', 'è', 's'
   ,'b','s', 'o', 'i', 'f']
```

Partie III Calcul de différentiels par calcul de plus courts chemins

Dans cette partie on souhaite exprimer le problème de calcul de distance d'édition comme un problème de calcul de plus court chemin dans un graphe orienté pondéré. Pour deux textes $\texttt{texte}_1$ et $\texttt{texte}_2$, on considère une grille de dimension $(\texttt{len}(\texttt{texte}_1) + 1) \times (\texttt{len}(\texttt{texte}_2) + 1))$ dont chaque cellule est un sommet du graphe. On appelle sommet un couple (i, j) tel que $0 \leq i \leq \texttt{len}(\texttt{texte}_1)$ et $0 \leq j \leq \texttt{len}(\texttt{texte}_2)$. Chaque sommet (i, j) aura au plus trois arcs sortants vers des sommets parmi $(i + 1, j), (i, j + 1)$ et $(i + 1, j + 1)$. On appelle entrée du graphe le sommet $(0, 0)$ et sortie le sommet $(\texttt{len}(\texttt{texte}_1), \texttt{len}(\texttt{texte}_2))$.

La FIGURE 4 présente la matrice de distance d'édition pour $\texttt{texte}_1$ = `['b', ' i', 'e ', 'n']` et $\texttt{texte}_2$ = `['b','o','n','n','e']`, ainsi que le graphe associé, sans les poids des arcs.

Le graphe ne sera jamais explicitement représenté, mais nous sommes en mesure de **calculer** l'ensemble des arcs sortants de chaque sommet.

Q14. Écrire une fonction `successeurs(texte1, texte2, sommet)` qui renvoie une liste de couples `(voisin, distance)`, de taille au plus 3, représentant les sommets destinations des arcs sortant du sommet `sommet`, avec les poids associés.

L'existence et la pondération des arcs devra permettre d'assurer la correspondance suivante entre le graphe et la matrice de distance d'édition de $\texttt{texte}_1$ et $\texttt{texte}_2$: pour tout sommet (i, j) du graphe, $M[i][j]$ coïncide avec la longueur d'un plus court chemin de $(0, 0)$ à (i, j).

Démontrer cette propriété avec une récurrence.

		b	o	n	n	e
	0	1	2	3	4	5
b	1	0	1	2	3	4
i	2	1	2	3	4	5
e	3	2	3	4	5	4
n	4	3	4	3	4	5

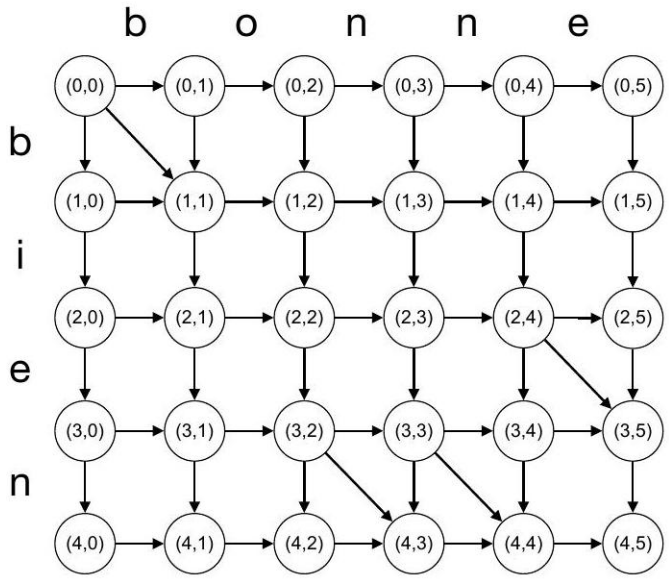


FIGURE 4 - Exemple de matrice de distance d'édition et de graphe associé (les poids des arcs ont été volontairement omis).

Exemple (les poids des arcs sont ici remplacés par ...)

```
>>> texte1 = ['b', 'i', 'e', 'n']
>>> texte2 = ['b', 'o', 'n', 'n', 'e']
>>> successeurs(texte1, texte2, (2,4))
[((3, 4), ...), ((2, 5), ...), ((3, 5), ...)]
```

Pour calculer un plus court chemin, on peut utiliser une variante l'algorithme de Dijkstra, présentée dans la FIGURE 5. Il s'appuie sur une structure de données de file de priorité sur les sommets (i, j) du graphe, dont on ne précise pas l'implémentation mais dont on précise ici la complexité des différentes opérations élémentaires.

- La fonction `vide()` construit une file vide (de cardinal 0) en $\mathcal{O}(1)$.
- La fonction `est_vide(file)` teste si la file `file` est vide en $\mathcal{O}(1)$.
- La fonction `extraire_min(file)` supprime l'élément de priorité minimale dans la file `file` et le renvoie. En cas d'égalité de priorités, elle renvoie le sommet (i, j) le plus petit pour l'ordre lexicographique [5] parmi les sommets de priorité minimale. Sa complexité est en $\mathcal{O}(\log(\text{cardinal}(\texttt{file})))$.
- La fonction `ajoute(file, sommet, priorité)` ajoute à la file `file` un sommet `sommet` avec une priorité `priorité`. L'opération augmente de 1 le cardinal de la file si le sommet n'est pas déjà présent avec cette priorité. Sa complexité est en $\mathcal{O}(\log(\text{cardinal}(\texttt{file})))$.

```python
def dijkstra(texte1, texte2):
    entrée = (0, 0)
    sortie = (len(texte1), len(texte2))
    file = vide()
    dist = {}
    vue = {}
    horloge = 0
    ajoute(file, entrée, 0)
    dist[entrée] = 0
    while not est_vide(file):
        sommet = extraire_min(file)
        if not sommet in vue:
            vue[sommet] = horloge
            horloge +=1
            if sommet == sortie:
                dist_final = {sommet: dist[sommet] for sommet in vue}
                return dist_final
            for voisin, distance in successeurs(texte1, texte2, sommet):
                d = dist[sommet] + distance
                if not voisin in dist or d < dist[voisin]:
                    dist[voisin] = d
                    ajoute(file, voisin, d)
    assert False
```

FIGURE 5 - Une variante de l'algorithme de Dijkstra.

Q15. En vous appuyant sur les propriétés de l'algorithme de Dijkstra vues en cours, expliquer pourquoi l'utilisation de la fonction `dijkstra` permet de calculer la distance d'édition entre $\texttt{texte}_1$ et $\texttt{texte}_2$. Préciser ce que contient le dictionnaire `dist_final` renvoyé, en caractérisant soigneusement l'ensemble des clés de ce dictionnaire.

Q16. Donner la complexité de la fonction `dijkstra` et commenter son intérêt par rapport à l'algorithme de programmation dynamique de la partie II.

On s'intéresse maintenant à l'algorithme A^*, présenté dans la FIGURE 6. Il s'appuie sur une fonction heuristique `h` qui estime la distance de chaque sommet à la sortie du graphe. On admet que cet algorithme renvoie un dictionnaire `dist_final` tel que `dist_final[sortie]` est la longueur d'un plus court chemin de l'entrée à la sortie du graphe, si la fonction heuristique `h` utilisée est admissible, c'est à dire si pour tout sommet `s` du graphe, $\texttt{h(texte}_1\texttt{, texte}_2\texttt{, s)}$ est inférieure ou égale à la longueur pondérée d'un plus court chemin de `s` jusqu'à la sortie du graphe.

Q17. Donner une fonction `h` qui satisfait cette hypothèse, avec une complexité en $\mathcal{O}(1)$, et qui permet un gain de temps de calcul (vis-à-vis du nombre de sommets extraits de la file avant de rencontrer la sortie) sur l'exemple $\texttt{texte}_1 = \texttt{['A', 'B', 'C']}$, $\texttt{texte}_2 = \texttt{['B', 'X']}$. Justifier en comparant les dictionnaires `dist_final` renvoyés par les deux algorithmes sur cet exemple.

```python
def astar(texte1, texte2):
    entrée = (0, 0)
    sortie = (len(texte1), len(texte2))
    file = vide()
    dist = {}
    vue = {}
    horloge = 0
    ajoute(file, entrée, 0)
    dist[entrée] = 0
    while not est_vide(file):
        sommet = extraire_min(file)
        vue[sommet] = horloge
        horloge += 1
        if sommet == sortie:
            dist_final = {sommet: dist[sommet] for sommet in vue}
            return dist_final
        for voisin, distance in successeurs(texte1, texte2, sommet):
            d = dist[sommet] + distance
            if not voisin in dist or d < dist[voisin]:
                dist[voisin] = d
                ajoute(file, voisin, d + h(texte1, texte2, voisin))
    assert False
```

FIGURE 6 - Algorithme A^*.