Algorithmique
TP 6 : Programmation Dynamique

L'exercice de ce TP doit être déposé sur l'espace "Rendu TP6".

1 Alignement de séquences

Distance d'alignement. Supposons que nous ayons deux séquences $a[1..n]$ et $b[1..m]$ et une fonction $c(x, y)$ qui nous dit combien « coûte » le remplacement du symbole x par y : par exemple $c(x, y) = 0$ si $x = y$ et $c(x, y) = 1$ si $x \neq y$.
Par exemple, voici deux alignements de « chat » et « chien » (on utilisera le symbole '-' pour dénoter un caractère espace) :

```
1)  ch-at       2)  c-----hat
    chien           -chien---
```

Le coût du premier est 3 (1 pour l'espace, 1 pour le remplacement a-e et 1 pour le remplacement t-n). Le deuxième coûte 9 (pour 9 insertions d'espaces, qui servent commes des « jokers »).
Ce coût est appelé *distance d'alignement*.

Problème de l'alignement optimal. Pour ce TP nous allons résoudre le problème algorithmique suivant : trouver la plus petite distance d'alignement entre les deux séquences $a[1..n]$ et $b[1..m]$, et produire un alignement atteignant cette distance.
Ce problème peut être résolu en utilisant l'algorithme de Needleman-Wunsch :

- On introduit une matrice $D[0..n, 0..m]$ telle que $D_{i,j}$ est la distance d'alignement entre les séquences $a[1..i]$ et $b[1..j]$.
- Le cas de base $D_{0,j}$ (quand la première séquence est vide) est trivial : pour aligner une séquence vide avec la séquence $b[1..j]$ qui a j éléments, on doit insérer j espaces, ainsi $D_{0,j} = j$. De manière similaire $D_{i,0} = i$.
- Supposons maintenant que i et j sont non nuls, et qu'on a déjà calculé les distances pour les séquences plus courtes. Pour analyser le coût d'alignement $D_{i,j}$ de $a[1..i]$ et $b[1..j]$, on regarde les derniers symboles a_i et b_j.

  On a alors trois possibilités :

  - ↖ On aligne a_i et b_j, ce qui coûte $c(a_i, b_j)$. Il faut aussi aligner toutes les lettres précédentes, ce qui donne le coût $D_{i,j} = D_{i-1,j-1} + c(a_i, b_j)$.
  - ← On insère un espace dans la première séquence, en face de b_j, et on aligne toutes les lettres précédentes : le coût est $D_{i,j} = D_{i,j-1} + 1$.
  - ↑ On insère un espace dans la deuxième séquence, en face de a_i, et on aligne toutes les lettres précédentes : le coût est $D_{i,j} = D_{i-1,j} + 1$.

  Pour avoir un alignement optimal on choisit la possibilité qui donne le coût minimal : il faut donc, à chaque étape, inscrire dans la matrice le minimum de ces trois valeurs.

2 Distance d'alignement

Dans cet exercice, vous implémenterez le calcul de la distance entre deux mots en utilisant l'algorithme de programmation dynamique qu'on vient de décrire.
Le pseudocode est le suivant :

```
int distance(a[1..n], b([1..m))
    // D[0..n, 0..m] = matrice de coûts
    // Initialisation
    pour i de 0 a n
        D[i, 0]= i
    pour j de 1 a m
        D[0, j]=j

    // Boucle principale
    pour i de 1 a n
        pour j de 1 a m
            D[i, j] = min(D[i-1, j-1] + c(a[i-1], b[j-1]),
                          D[i, j-1] + 1,
                          D[i-1, j] + 1)

    retourner D[n,m]
```

Dans le fichier `Distance.java` complétez l'implémentation de la classe `Distance` :

1. Définir deux attributs `String` pour les 2 mots à comparer.
2. Définir une matrice (tableau bi-dimensionnel) d'**`int`** contenant les coûts calculés par l'algorithme de programmation dynamique.
3. Définir un constructeur prenant deux `String` en paramètres, initialisant les attributs `String` à partir de ceux-ci et créant la matrice de coûts.
4. Compléter l'implémentation de la méthode **`public int`** `cost(`**`char`** `a, `**`char`** `b)` qui calcule le coût du remplacement du symbole a par b : on considère qu'insérer un espace ou remplacer un symbole coûte 1, ce qui revient à calculer la distance d'édition classique de Levenshtein.
5. Coder la méthode **`public void`** `computeCosts()` qui calcule la matrice de coûts (rappel : pour lire le $i$ième caractère d'un `String`, on peut utiliser la méthode `charAt`).
6. Coder la méthode **`public int`** `distance()` qui retourne la distance d'alignement entre les mots initialisés dans le constructeur.

3 Calcul d'un alignement optimal ✍

On veut maintenant afficher un alignement réalisant le coût minimal calculé dans l'exercice précédent.
Pour récupérer l'alignement optimal (et pas seulement son coût), on va mémoriser pour chaque élément $D_{i,j}$, quel cas a donné le minimum (on les dénotera $\nwarrow$, $\leftarrow$ ou $\uparrow$). Quand le calcul de la matrice D est terminé, on se rend à la case $D_{n,m}$ et on construit l'alignement de droite à gauche en remontant les flèches. Par exemple, si la flèche dans la dernière case est $\nwarrow$, on aligne a_n avec b_m et on monte vers la case en haut à gauche $D_{n-1,m-1}$ pour continuer de la même façon.
L'algorithme précédent devient donc :

```
int distance(a[1..n], b([1..m))
    // D[0..n, 0..m] = matrice de coûts
    // F[0..n, 0..m] = matrice de flèches
    // Initialisation
    pour i de 0 a n
        D[i, 0]= i; F[i, 0] = ↑
    pour j de 1 a m
        D[0, j]=j; F[0, j] = ←

    // Boucle principale
    pour i de 1 a n
        pour j de 1 a m
            x = D[i-1, j-1] + c(a[i-1], b[j-1])
            y = D[i, j-1] + 1
            z = D[i-1, j] + 1
            si x <= y et x <= z
                D[i, j] = x; F[i, j] = ↖
            sinon si y <= x et y <= z
                D[i, j] = y; F[i, j] = ←
            sinon
                D[i, j] = z; F[i, j] = ↑

    retourner D[n,m]
```

Complétez la classe de l'exercice précédent, avec :

1. L'ajout du type énuméré `Fleche` pour représenter les 3 flèches du pseudo-code.

   Vous pouvez juste recopier la définition suivante :

   ```
   public enum Fleche { HAUT, HAUT_GAUCHE, GAUCHE }
   ```

2. Un attribut supplémentaire (matrice à valeurs dans `Fleche`) indiquant, pour chaque case, si celle-ci reprend le score de la case de gauche ($\leftarrow$), celui de la case du dessus ($\uparrow$) ou celui de la case située en haut à gauche ($\nwarrow$).

3. La création de la matrice supplémentaire dans le constructeur.

4. Des instructions supplémentaires dans `computeCosts`, pour remplir le contenu de cette matrice en même temps qu'on calcule les coûts.

5. Une méthode `public String alignement()` retournant une représentation des deux mots alignés. Pour ce faire, il faut partir de la fin des mots et « remonter » la matrice en suivant les flèches.