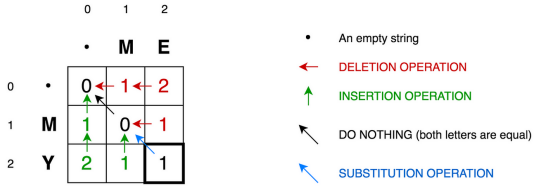
Chapitre 4

DISTANCE D'ÉDITION (LEVENSHTEIN)

Cette séance traite un **exemple classique de la programmation dynamique** clairement indiqué comme « bon a aborder » au programme d'informatique de seconde année. Il s'agit de la distance d'édition, une **distance entre chaînes de caractères**, appelée également distance de Levenshtein, du nom du scientifique russe Vladimir Iossifovitch Levenshtein (1935-2017) dont les travaux portent en grande partie sur la théorie des codes. La distance d'édition a de **nombreuses applications**, notamment en linguistique (correcteur orthographique, etc.) et en génétique (l'ADN étant modélisé comme une chaînes de caractères sur l'alphabet $\{A, C, G, T\}$).

Il est très possible qu'un sujet de **concours** (X-ENS PSI 2023 par exemple) s'inspire de ce type d'exemple classique d'algorithme pour vous demander de **traiter un cas similaire** voire identique. Théoriquement vous n'avez pas à connaitre quoi que ce soit par cœur, car tout le contexte et les idées seront normalement ré-expliqués dans l'énoncé (plus ou moins précisément donc il faut déjà être familier avec le concept).



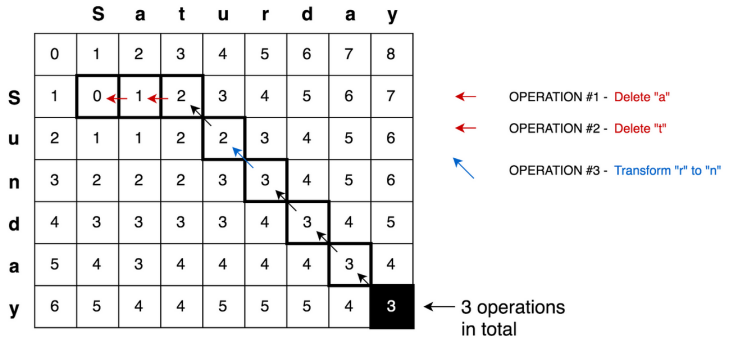


Table des matières

4 DISTANCE D'ÉDITION (LEVENSHTEIN) 1

I PRINCIPE DE LA DISTANCE D'ÉDITION DE LEVENSHTEIN 2

II EXERCICES : AUTOUR DE LA DISTANCE D'ÉDITION 3

I PRINCIPE DE LA DISTANCE D'ÉDITION DE LEVENSHTEIN

La distance d'édition, ou distance de Levenshtein, est une mesure de la similarité de deux chaînes de caractères : elle est égale au **nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer** pour passer d'une chaîne de caractères à une autre.

Par exemple, on peut passer du mot `Sarah` au mot `Natacha` en suivant les étapes suivantes :

— remplacement du `'S'` par un `'N'` : `Sarah` → `Narah` ;

— remplacement du `'r'` par un `'t'` : `Narah` → `Natah` ;

— insertion de la lettre `'c'` : `Natah` → `Natach` ;

— insertion de la lettre `'a'` : `Natach` → `Natacha` ;

$$\texttt{Sarah} \to \texttt{Narah} \to \texttt{Natah} \to \texttt{Natach} \to \texttt{Natacha}$$

donc la distance d'édition entre ces deux mots au plus de 4. On se convaincra qu'il n'est pas possible de faire mieux, la distance est donc égale à 4.

On peut calculer la distance d'édition entre deux mots $a = a_1a_2\cdots a_n$ et $b = b_1b_2\cdots b_p$ en généralisant le problème, c'est à dire en calculant successivement la distance d'édition $\mathrm{d}(i,j)$ entre les préfixes $a_1a_2\cdots a_i$ et $b_1b_2\cdots b_j$ pour $i \in [\![1,n]\!]$ et $j \in [\![1,p]\!]$.

Pour calculer $\mathrm{d}(i,j)$ on se concentre sur le dernier caractère de chacun des deux préfixes. Dans le chemin reliant de manière optimale $a_1a_2\cdots a_i$ et $b_1b_2\cdots b_j$, plusieurs cas de figure peuvent se rencontrer :

— a_i a été supprimé (**suppression**) ;

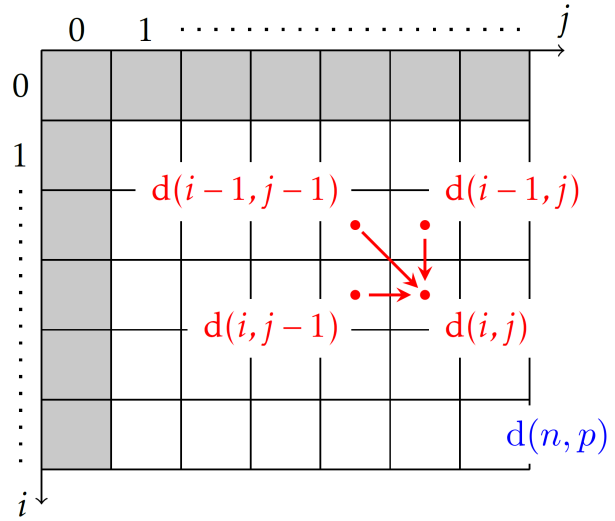
— b_j a été ajouté (**insertion**) ;

— a_i a été remplacé par b_j (**substitution**) ;

— $a_i = b_j$.

Les 4 cas ci-dessus vont mener à des relations de dépendance entre $\mathrm{d}(i,j)$ et les valeurs de $\mathrm{d}(i-1,j)$, $\mathrm{d}(i,j-1)$ et $\mathrm{d}(i-1,j-1)$. La formule exacte sera à déterminer dans les exercices ci-dessous.

Les conditions initiales sont données par les valeurs de $\mathrm{d}(i,0)$ et $\mathrm{d}(0,j)$ (distance d'édition à un mot « vide ») qui seront à déterminer. Ce qui conduit au schéma de dépendance représenté ci-dessous :



Exemple. Pour `mot1="Sarah"` et `mot2="Natacha"`.

On aurait `n=5` et `p=7`.

- $\mathrm{d}(0,0) = 0$ est la distance de `""` à `""`
- $\mathrm{d}(0,1) = 1$ est la distance de `""` à `"N"`
- $\mathrm{d}(1,0) = 1$ est la distance de `"S"` à `""`
- $\mathrm{d}(1,1) = 1$ est la distance de `"S"` à `"N"`
- $\mathrm{d}(0,2) = 2$ est la distance de `""` à `"Na"`
- $\mathrm{d}(2,0) = 2$ est la distance de `"Sa"` à `""`
- $\mathrm{d}(1,2) = 2$ est la distance de `"S"` à `"Na"`
- $\mathrm{d}(2,1) = 2$ est la distance de `"Sa"` à `"N"`
- $\mathrm{d}(2,2) = 1$ est la distance de `"Sa"` à `"Na"`

Et ainsi de suite...

Une fonction récursive classique aurait-elle une complexité acceptable ? Pourquoi ?
Si on choisit de remplir un tableau, quelle taille devra avoir ce tableau ?

II EXERCICES : AUTOUR DE LA DISTANCE D'ÉDITION

Exercice 1 (**Distance d'édition de Levenshtein**)**.**

Considérons deux mots `mot1` et `mot2`. On pose `n=len(mot1)` et `p=len(mot2)`.

On cherche la distance de Levensthein entre `mot1` et `mot2`, soit le nombre minimum d'opérations qu'il faut faire pour passer de `mot1` à `mot2`.

Le but est donc de trouver un algorithme qui calcule cette distance. On stocke les valeurs de $\mathrm{d}(i,j)$ dans un tableau bidimensionnel `D` de taille $n+1 \times p+1$ tel que `D[i][j]` soit la valeur de $\mathrm{d}(i,j)$, la distance de Levensthein entre les mots `mot1[:i]` et `mot2[:j]`.

1. Pour `i` dans $[\![0,n]\!]$ et `j` dans $[\![0,p]\!]$, que valent `D[0][j]` et `D[i][0]` ?

   .....

2. Une fois qu'on aura correctement rempli `D`, où exactement trouvera-t-on la distance de Levensthein entre `mot1` et `mot2` ?

   .....

On prend maintenant `i` dans $[\![1,n]\!]$ et `j` dans $[\![1,p]\!]$.

3. Si `mot1[i-1]=mot2[j-1]`, exprimer `D[i][j]` en fonction de `D[i-1][j-1]`.

   `D[i][j]=`.....

4. Si `mot1[i-1]`$\neq$`mot2[j-1]`, alors pour aller de `mot1[:i]` à `mot2[:j]` il faudra changer la dernière lettre, il y a donc trois possibilités. Pour chaque cas, exprimer `D[i][j]` en fonction de `D[i-1][j-1]`, `D[i-1][j]` et/ou `D[i][j-1]` :
   - On rajoute la dernière lettre de `mot2[:j]`

     `D[i][j]=`.....
   - On supprime la dernière lettre de `mot1[:i]`

     `D[i][j]=`.....
   - On modifie la dernière lettre de `mot1[:i]` pour qu'elle vaille la dernière lettre de `mot2[:j]`

     `D[i][j]=`.....

   Trouver alors une formule de `D[i][j]` valable dans ces trois cas, en se rappelant que la distance correspond au plus court chemin de `mot1` à `mot2` :

   `D[i][j]=`.....

5. Programmer une fonction `distance(mot1,mot2)` par une méthode de programmation dynamique de bas en haut. Quelques indications :
   - créer d'abord la liste bidimensionnelle `D` évoquée plus haut initialement remplie de 0,
   - remplir les valeurs de `D[i][j]` si `i` ou `j` vaut 0,
   - remplir ensuite les valeurs de `D[i][j]` pour `i`$\geqslant 1$ et `j`$\geqslant 1$ suivant la valeur de récurrence trouvée (attention à bien distinguer les cas).
6. Tester avec `distance("Sarah","Natacha")`, puis avec les mots `informatique` et `affirmative`.
7. Quelle est la complexité temporelle et spatiale de la fonction `distance` en fonction de `n` et `p` ?
8. Récupérer la liste des mots français du TP3, puis écrire une fonction `ListeMotsPlusProche(maux)` qui étant donné une chaîne de caractères, notée `maux` (« mot » mal orthographié), renvoie la liste des mots les plus proches (pour la distance de Levensthein) de `maux` dans la liste des mots français.
9. Votre professeur d'imffaurmathiqe étant très mauvais en orthographe, trouver la liste des mots français les plus proche de ce mot mal orthographié. Si vous êtes arrivés là, félicitations, vous venez de programmer un correcteur orthographique !

Exercice 2 (**Distance de Levenshtein avec mémoïsation**).

Écrire une version récursive avec mémoïsation de la distance de Levenstein.

Exercice 3 (**Reconstruction du chemin minimum entre deux mots**).

On souhaite écrire une fonction récursive `Chemin(mot1,mot2)` qui, étant donnés deux mots, donne un chemin minimal de l'un à l'autre sous forme de liste comme `[Sarah,Narah,Natah,Natach,Natacha]`.

Notez qu'il n'y a pas nécessairement unicité du chemin minimal, plusieurs chemins pouvant avoir la même longueur. On se contentera de calculer l'un des chemins minimaux.

1. Modifier votre fonction `distance(mot1,mot2)` en une fonction `distance2(mot1,mot2)` qui renvoie le tableau bidimensionnel `D` (on en aura besoin à la fin de cet exercice).
2. Donner le chemin minimal dans le cas où `mot1` est vide (idem pour `mot2`).

   .....

3. On se place dans le cas où `mot1` et `mot2` finissent par la même lettre.
   Exprimer `Chemin(mot1,mot2)` en fonction de `Chemin(mot1[:n-1],mot2[:p-1])`.

   .....

4. On se place dans le cas où la dernière lettre `mot1` est différente de celle de `mot2`. Dans ce cas, la dernière étape sera soit un ajout, soit une suppression soit une modification de la dernière lettre.
   Grâce à la liste `D`, distinguez les cas, pour savoir s'il faut construire `Chemin(mot1,mot2)`
   - à partir de `Chemin(mot1[:n-1],mot2)` ;
   - ou à partir de `Chemin(mot1,mot2[:p-1])` ;
   - ou bien à partir de `Chemin(mot1[:n-1],mot2[:p-1])`.

   .....

   .....

   .....

5. Implanter la fonction `Chemin` en entier, et la tester avec `Chemin("informatique","affirmative")` puis `Chemin("Sarah","Natacha")`.

Exercice 4 (**Plus longue sous-séquence commune**).

Finissez le TP de la séance précédente, en particulier l'exercice sur la plus grande sous-séquence commune entre deux mots (cet exemple est lui aussi explicitement mentionné par le programme officiel de PSI).