Programmation dynamique

La quasi-totalité des Smartphones actuels sont munis d'une fonction de correction automatique (parfois un peu trop) pour permettre aux usagers de rédiger des messages rapides sans avoir trop à se soucier d'être particulièrement vigilants pour l'orthographe. Lorsqu'un message est rédigé rapidement, les principales sources de coquilles sont les suivantes :

- oubli d'un caractère ;
- utilisation d'un mauvais caractère ;
- ajout d'un caractère incorrect (pour ceux qui ont des gros doigts).

Ainsi, on pourra se retrouver à écrire `endamrphismle` au lieu de `endomorphisme` en cumulant les trois coquilles ci-dessus. Pendant la composition d'un mot, les outils numériques proposent donc à la volée une liste de mots du dictionnaire parmi les plus proches de celui qui est en train d'être écrire pour corriger les erreurs ou même compléter automatiquement la fin du mot.

Dans ce TP, on va s'intéresser à une méthode pour déterminer parmi un ensemble E de mots celui ou ceux qui sont les plus proches d'un mot u. Cette méthode est basée sur la notion de distance de Levenstein, aussi appelée distance d'édition, qui compte le nombre minimal d'opérations élémentaires pour passer d'un mot u à un mot v.

1 Distance de Levenstein

Dans toute cette partie, on considère deux mots u et v comportant respectivement p et q lettres. En Python, ces deux mots seront donnés sous la forme d'une chaine de caractères. On rappelle que les chaînes fonctionnent comme les listes : on accède au i-ième caractère par la syntaxe `u[i]` et la première lettre est d'indice 0. Le mot vide (qui ne comporte aucune lettre) s'écrit `''` ou `""`. On le notera ϵ par la suite.

On définit trois types d'opérations élémentaires sur une chaine de caractère u :

- `ajout(u,i,a)` qui ajoute un caractère à la i-ème position du mot u. Par exemple, `ajout('toto', 1, 'z')` renvoie `'azjout'`.
- `supprime(u,i)` qui supprime le i-ième caractère de u. Ainsi, l'instruction `supprime('toto',1)` renvoie `'tto'`.
- Enfin, `remplace(u,i,a)` qui remplace le i-ième caractère de u par la lettre `a` de sorte que `remplace('toto',3,'u')` renvoie `'totu'`.

Note : l'implémentation de ces fonctions n'est pas nécessaire dans ce TP.

On notera qu'on aurait pu se passer de la dernière opération qui s'obtient en composant les deux précédentes, mais en informatique, il est moins couteux de remplacer un caractère que de le supprimer.

La distance de Levenstein $d(u,v)$ entre u et v est alors définie comme le nombre minimal d'opérations élémentaires pour passer de u à v.

1. Justifier que $d(u,\epsilon) = p$ pour tout mot u comportant p lettres.
2. Justifier que $d(u,v) = d(v,u)$ pour tous mots u et v.

Pour tout $i \in [\![0;p]\!]$, on note u_i le mot constitué des i premières lettres de u. On définit de même v_j pour tout $j \in [\![0;q]\!]$ et on pose

$$\forall i,j \in [\![0;p]\!] \times [\![0;q]\!], \qquad d_{i,j} = d(u_i, v_j)$$

On notera donc que $d(u,v) = d_{p,q}$. Cette quantité va s'obtenir en calculant la matrice $D = (d_{i,j})_{i,j \in [\![0;p]\!] \times [\![0;q]\!]}$.

3. **Résultat principal :** Justifier que pour tous $(i,j) \in [\![0;p-1]\!] \times [\![0;q-1]\!]$

$$d_{i+1,j+1} = \begin{cases} d_{i,j} & \text{si } u[i] = v[j] \\ 1 + \min\{d_{i+1,j}, d_{i,j+1}, d_{i,j}\} & \text{sinon} \end{cases}$$

4. En déduire une fonction `distance` qui prend en argument deux mots u et v et renvoie la distance de Levenstein entre les deux mots u et v. On utilisera la méthode suivante :
   - Initialiser une matrice `D` sous la forme d'une liste de p listes de 0 de longueur q. La case $D[i][j]$ contiendra à terme $d_{i,j}$. On pourra éventuellement utiliser un dictionnaire (initialement vide) pour ceux qui préfèrent.
   - Remplir la première colonne et la première ligne de D en utilisant la question 1.
   - Remplir le reste de D ligne par ligne (ou colonne par colonne) en utilisant la question 3.

   On testera le résultat sur plusieurs couples de mots de votre choix.

2 Tris efficaces

Dans cette partie, on dispose d'un ensemble E de mots et d'un mot u. On cherche dans un premier temps à trier efficacement l'ensemble E par ordre croissant de la distance à u. L'opération restant coûteuse, on cherchera ensuite plus simplement à extraire de E les k éléments à distance la plus faible de u, l'entier k étant donné en argument.

2.1 Tri fusion

Le tri fusion est un algorithme efficace qui permet de trier les éléments d'une liste de n éléments en $O(n \ln n)$ opérations élémentaires. Il est relativement optimal dans la mesure où on peut démontrer que tout algorithme de tri n'utilisant que des comparaisons entre les éléments de la liste utilise systématiquement dans le pire cas au moins $O(n \ln n)$ opérations. Il est basé sur la stratégie « diviser pour régner » (qui regroupe une vaste classe d'algorithmes). Pour trier une liste de longueur n :

- Si $n \leq 1$, il n'y a rien à faire, on renvoie la liste sans la modifier.
- Si $n \geq 2$, on coupe la liste en deux, on trie les deux morceaux, puis on réalise une fusion pour reconstituer la liste triée à partir des deux sous-listes triées.

Pour réaliser l'opération de fusion de deux listes `L1` et `L2` triée et de longueur n_1 et n_2,

- on crée une nouvelle liste `L` (vide, ou de longueur $n_1 + n_2$) dans laquelle on va ajouter successivement les éléments des deux listes par ordre croissant.
- on utilise deux indices `i` et `j` pour parcourir simultanément `L1` et `L2`. À chaque instant, on compare `L1[i]` et `L2[j]` et on ajoute le minimum dans `L`. Après quoi, on augmente `i` ou `j` de 1 suivant l'élément ajouté.
- Dès que l'on arrive à la fin d'une liste, on ajoute les éléments restants de l'autre liste les uns après les autres.

5. Ecrire la fonction `fusion` décrite ci-dessus. Elle prendra en argument les deux listes `L1` et `L2` ainsi que la fonction de comparaison `compare`. On n'utilisera donc pas l'opérateur `<` pour comparer deux éléments `x` et `y` mais l'instruction `compare x y` qui renvoie le booléen `True` si et seulement si x est plus petit que y au sens de la comparaison utilisée.
6. En déduire la fonction `tri_fusion` qui prend en argument une liste `L` et la trie par ordre croissant selon une relation de comparaison là encore donnée en argument sous la forme d'une fonction `compare`.
7. **Application :** Etant donné un ensemble de mots E représentée en python sous la forme d'une liste de chaînes de caractères et un mot u, écrire une fonction `dico_trié` qui renvoie une liste contenant les mêmes mots que E mais rangé par ordre croissant de distance de Levenstein à u.

2.2 Extraction des k minimums

Les algorithmes de correction automatique proposent en général un faible nombre de corrections (trois ou quatre). Il n'est donc pas indispensable de trier l'intégralité du dictionnaire mais seulement d'en extraire les plus petits éléments.

8. Ecrire une fonction qui insère un élément v dans une liste `L` triée de longueur inférieur ou égale à k. L'élément v est inséré à la bonne place en décalant tous les éléments suivants et le dernier élément est supprimé du tableau si sa longueur dépasse k après l'insertion. A nouveau, la fonction de comparaison `compare` sera fournie en argument.
9. En déduire une fonction `k_plus_proches` qui prend en argument l'ensemble E (sous la forme d'une liste de chaînes de caractères) et un mot u et renvoie les k mots de E les plus proches de u.

3 Pour les plus motivés

La fonction `distance` de la première partie renvoie la distance entre deux mots u et v mais ne précise pas par quelles opérations on peut passer de u à v. On veut maintenant écrire une fonction qui renvoie la suite de ces opérations. On reprend pour cela l'analyse amenant à la formule de la question 3 : on va déterminer les opérations pour passer de $u = u_p$ à $v = v_q$ par récurrence :

- Si `u[i] = v[j]`, on passe de u_{i+1} à v_{j+1} de la même manière que l'on passe de u_i à v_j.
- Sinon, on distingue les cas du minimum :
  - si $d_{i+1,j+1} = d_{i,j}$, on passe de u_{i+1} à v_{j+1} en effectuant les opérations pour passer de u_i à v_j, puis en remplaçant la lettre `u[i]` par `v[j]` ;
  - si $d_{i+1,j+1} = d_{i+1,j}$, on passe de u_{i+1} à v_{j+1} en effectuant les opérations pour passer u_{i+1} par v_j, puis ajoutant la lettre `v[j]` en fin de mot ;
  - le dernier cas est symétrique.

On peut ainsi écrire une fonction récursive qui liste/affiche les opérations à effectuer pour passer de u_i à v_j et qui s'arrête lorsque $i = j = 0$ et l'appliquer au couple (u_p, v_q). On commence pour cela par calculer les valeurs $(d_{i,j})$ comme précédemment avant d'exécuter la fonction suivant le principe récursif détaillé ci-dessus.

10. Ecrire une fonction `transformation` qui prend en argument les mots u et v et affiche la liste des opérations pour passer de u à v.