TP Programation dynamique.

Avant propos : je dois retaper le TP en urgence. La mise en page est plus austère que d'habitude… Sachez que dans cette mise ne page, les guillemets ‘ ainsi que leur version triple “‘ servent à délimiter du code.

Dans ce TP, on travaillera en C.

Problème : la distance d'édition

Type char*

Un *mot* est une suite de lettres. En C, une suite de lettres est de type `char*` :

```
char* demo = "La programmation dynamique, c'est une recurrence intelligente";
```

Afin de limiter les problèmes, on se restreint aux lettres du type `char`, aka l'ASCII. On précise qu'en C, un `char*` comme ci-dessus est un pointeur vers une zone de la mémoire (d'où son type). Dans cette zone se trouvent les lettres les unes après les autres. Enfin, il faut marquer la fin du mot (sinon on ne sait pas quand on atteint la fin de la zone) : après toutes ces lettres, le mot est terminé par un caractère spécial `\0`.

Ex : le mot “elephant” est correspond à la zone mémoire :

```
| e | l | e | p | h | a | n | t | \0
```

La longueur d'un mot est son nombre de lettres (`\0`exclu). La fonction `strlen`, rangée dans `string.h`, permet de calculer la longueur.

Distance d'édition

L'enjeu de ce TP est de calculer la distance d'édition. C'est un problème avec de nombreuses applications, la plus célèbre étant le calcul du nombre de mutations nécessaires pour qu'un allèle d'un gêne en soit muté en un autre.

On considère trois façons de modifier/muter un mot : - délétion : on supprime une lettre quelque part dans le mot - addition : on ajoute une lettre quelque part dans le mot - substituion : une lettre du mot est remplacé par une autre

Par exemple, le mot “elements” peut-être transformé en “elephant” ainsi : elements -> elepments -> elephents -> elephants -> elephants

La *distance d'édition* entre deux mots u et v, notée d(u,v), est le nombre minimal d'opérations à effectuer pour transformer u en v. On l'appelle également *distance de Levenshtein*.

NB : c'est un grand classique de concours. Par exemple, Centrale ITC 2023.

On admet qu'il s'agit d'une distance, c'est à dire que pour tous u,v,w : - d(u,v) = d(v,u) : essayer de transformer u en v est équivalent à v en u - d(u,v) >= 0, avec le cas d'égalité atteint SSI u=v - d(u,w) >= d(u,v) + d(v,w) : forcer à

passer par une étape intermédiaire précise dans la transformation ne peut pas raccourcir la distance. Cette propriété s'appelle l'inégalité triangulaire.

L'inégalité triangulaire est intuitive : pour aller de Poitiers à Toulouse, forcer une ville intermédiaire soit rallonge (si l'intermédiaire fait faire un détour), soit laisse la distance inchangée (si l'intermédiaire est sur le bon chemin).

1) Calculer la distance d'édition entre "plume" et "palme".

Résolution par récurrence

La méthode par récurrence va procéder en parcourant les mots de droite à gauche. En fait, elle cherche à d'abord rendre égales les fins des mots, puis continuer récursivement pour rendre égaux les débuts. On ne va donc travailler *que* sur la fin des mots. Il y a cinq cas possibles : - un des deux mots est vide : dans ce cas il faut insérer toutes les lettres de l'autre. Fin. - la dernière lettre de chaque mot est égale : dans ce cas, rien à faire : on passe aux lettres suivantes. - sinon, on peut : - supprimer la lettre de fin du premier mot - ajouter la lettre de fin du second mot à la fin du premier - transformer la lettre du premier mot en la lettre du second On doit alors tester ces trois possibilités et voir laquelle mène in fine à la plus petite distance.

On admet que cette énumération est exhaustive, et que cela permet de calculer la distance de Levnshtein (bonus : le prouver).

Soient u et v deux mots. Notons lev(i,j) la distance entre u[0…i[et v[0…j[. L'énumération précédente se traduit alors en la formule de récurrence suivante :

lev(i,j) = - max(i,j) lorsque min(i,j) = 0 - lev(i-1,j-1) lorsque `u[i-1]` = `v[j-1]` - 1 + max(lev(i-1, j), lev(i, j-1), lev(i-1,j-1)) sinon

Le cas compliqué est de comprendre pourquoi l'ajout devient lev(i,j-1). L'idée est que l'on a transformé `u[0...i[` en `u[0...i[` suivi de `v[j-1]`. On est maintenant dans le cas où les dernières lettres des deux mots sont égales, et on doit donc égaler `u[0...i[` et `v[0...j-1[`.

(Me demander un schéma au tableau)

2) Vérifier à l'aide de cette formule votre hypothèse pour "plume" et "palme".
3) Coder une fonction récursive `int levenshtein_naif(char const* u, int i, char const* v, int j)`qui calcule la distance de levenshtein en appliquant la formule précédente.
4) On admet que la complexité de `levenshtein_naif` s'exprime de la forme C(i+j). Donnez la formule de récurrence vérifiée par C, et déduisez-en la complexité.

Programmation dynamique

5) Montrer sur un exemple que des sous-problèmes sont recalculés plusieurs fois.

Topdown

6) Écrivez une version mémoïsée de `levnshtein_naif`. Elle sera donc consitutée de deux fonctions :

- `int lev_topdown_rec(char const* u, int i, char const* v, int j, int** mem)` . Cette fonction est « celle qui fait tout le travail » : elle suppose une mémoire `mem` déjà crée et initialisée, procède par récurrence en mémorisant les lev(x,y) dans `mem[x][y]`.
- `int levenshtein_topdown(char const* u, int i, char const* v, int j)`. Cette fonction crée et initialise la mémoire, puis appelle la précdente, en déduit la valeur à renvoyer, libère la mémoire, et renvoie la valeur.

7) Comparez sur des exemples la vitesse d'exécution des deux fonctions.
8) Calculer la complexité temporelle de cette version récursive.

Bottom up

9) Sur un schéma, illustrer l'ordre de remplissage des lignes/colonnes de `mem`.
10) En déduire `int levenshtein_bottomup(char const* u, int i, char const* v, int j)`, une version dérécursifiée de la précédente (plusieurs choix d'ordres de remplissage sont possibles : allez au plus simple).
11) Calculer la complexité temporelle et spatiale de la précédente.
12) Optimisez la précédente pour améliorer l'espace utilisé afin de créer `int levenshtein(char const* u, int i, char const* v, int j)`.

Suite de mutations

13) Reconstruisez la suite de transformations permettant de transformer u en , çad ne calculez pas uniquement la distance mais aussi les modifs à faire. Vous pouvez vous contenter d'afficher cette suite de transformations au lieu d'essayer de la renvoyer.

Pour occuper les plus rapides

14) Voici un problème qui peut se résoudre par récurrence (et peut-être s'accélérer par prog dyn) :

- Entrée : M une matrice de 0 et de 1
- Sortie : un plus grand bloc rectangulaire de M ne contenant que des 0. On le représentera par la donnée de son angle en haut à gauche et de ses dimensions verticales et horizontales.