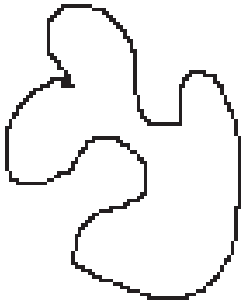
Récursivité et programmation dynamique

1 Traitement recursif d'une image

Le but de cet exercice est de remplir en vert une zone blanche de l'image, sans dépasser la ligne noire : remplir soit l'intérieur soit l'extérieur du contour.



Q1 : Écrire le programme qui charge l'image dans un tableau NumPy. Il affichera aussi l'image.

Q2 : Quel est le profil colorimétrique de l'image ? Quelles sont ses dimensions ?

Bien que l'image soit petite, il faut augmenter le niveau de récursion autorisé par Python : `sys.setrecursionlimit(10000)`

Q3 : Écrire une fonction récursive `remplissage` dont la finalité est de remplir une zone blanche, de proche en proche : Elle prend en argument les coordonnées d'un pixel puis :
- teste si le pixel est blanc,
- si oui : elle colorie ce pixel en vert, et s'appelle elle-même pour les quatres pixels adjacents.

Q4 : Tester votre fonction pour un pixel à l'intérieur du contour (position $80, 80$)

Q5 : Mettre en place un compteur pour déterminer combien d'appels récursifs ont été nécessaires.

2 Distance d'édition ou de *Levenshtein*

Lorsqu'un correcteur d'orthographe propose une correction, c'est-à-dire un nouveau mot proche de celui proposé par l'utilisateur, il choisit ce nouveau mot « proche » du mot mal orthographié. Il est nécessaire de quantifier la distance entre les deux mots.

Ceci est possible grâce à la distance de Levenshtein ou distance d'édition.

La distance de Levenshtein est une valeur donnant une mesure de la différence entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne `s1` à une chaîne `s2`.

La distance entre deux chaînes de caractères `s1` et `s2` est le nombre minimal d'opérations élémentaires permettant de transformer `s1` en `s2`, où l'on entend par opération élémentaire :
- la substitution d'un caratère par un autre,
- l'insertion d'un caractère,
- la suppression d'un caractère.

Quelques exemples : (on note $lev(s_1, s_2)$ La distance entre deux chaînes de caractères `s1` et `s2`)

- Pour tout chaîne `s` : $lev(s, s) = 0$
  (aucune opération n'est nécessaire pour transformer une chaîne en elle même !)
- $lev('lopin',' lapin') = 1$
  (on peut passer de `lopin` à `lapin` en substituant le `o` par un `a` (1 opération), et il n'est pas possible de faire mieux.)
- $lev('ACTGGT',' ACGGTA') = 2$
  (on peut passer de la première chaîne à la seconde par suppression du premier `T`, et insertion d'un `A` à la fin, et on vérifie qu'il n'est pas possible de faire mieux.

La distance d'édition vérifie la formule suivante, qui va permettre de la calculer de façon récursive :

$$lev(s_1, s_2) = \begin{cases} len(s_1) \text{ si } s_2 =' \ ' & (1) \\ len(s_2) \text{ si } s_1 =' \ ' & (2) \\ \text{lev}(s_1[1:], s_2[1:]) \text{ si } s_1[0] = s_2[0] & (3) \\ 1 + \min(\text{lev}(s_1[1:], s_2),\ \text{lev}(s_1, s_2[1:]),\ \text{lev}(s_1[1:], s_2[1:])) \text{ sinon} & (4) \end{cases}$$

- (1) : transformer s1 en la chaîne vide nécessite len(s1) suppressions.
- (2) : transformer la chaîne vide en s2 nécessite len(s2) insertions.
- (3) : si les premiers caractères de s1 et s2 correspondent, leur distance d'édition ne change pas si on les supprime.
- (4) : si les premiers caractères différent, c'est qu'il y a eu soit une suppression, soit une insertion, soit une substitution (correspondant aux trois arguments du min), et par conséquent un coût de +1 à payer.

2.1 Mise en place récursive

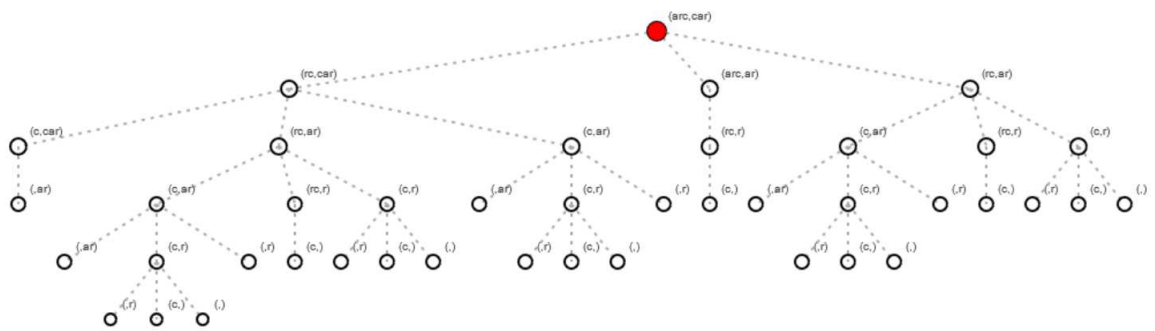
Q1 : Mettre en place le programme affichant la distance entre 2 chaines de caractères.

Q2 : Vérifier que la distance d'édition entre « arc » et « car » est de 2 ; celle entre « chien » et « niche » est de 4.

Q3 : Combien d'appel récursifs ont été nécessaires dans le dernier cas ?

2.2 Usage de la mémoïsation

La figure ci-dessous est une représentation des appels récursifs lors de la détermination de la distance entre entre « arc » et « car ».



On constate en regardant l'arbre que des appels récursifs qu'il y a de nombreux chevauchements de sous-problèmes (la fonction est appelée plusieurs fois avec les mêmes paramètres). On peut pour y remédier utiliser une stratégie descendante avec mémoïsation. En effet, la résolution fait appel à $lev(s_1[i:], s_2[j:])$ un grand nombre de fois.

Pour mémoriser ces informations, on va utiliser un tableau de taille $(len(s_1)+1) \times (len(s_2)+1)$ dont l'élément d'indices (i, j) contient la valeur de $lev(s_1[i:], s_2[j:])$ si celle-ci a déjà été calculée.

Q1 : Sur feuille : quelle sera l'allure du tableau lors de la détermination de la distance entre entre « arc » et « car » ?

Q2 : Ecrire un nouveau programme, complétant le premier programme par la mémoïsation nécessaire.

Q3 : Combien d'appel récursifs ont été nécessaires pour la détermination de la distance d'édition entre « chien » et « niche » ?

2.3 Approche ascendante

Chaque case du tableau ne dépend que des valeurs situées à droite, en dessous, et en diagonale en dessous à droite. Il est donc possible de résoudre le problème par un algorithme itératif ascendant : une nouvelle valeur est déterminée à partir des trois cases (à droite, en dessous, et en diagonale en dessous à droite) :

$$t[i][j] \leftarrow \begin{cases} t[i+1][j+1] & \text{si } s_1[i] = s_2[j] \\ t[i][j] \leftarrow 1 + \min(t[i+1][j], t[i][j+1], t[i+1][j+1]) & \text{sinon} \end{cases}$$

à l'initialisation, la cas en base à droite est à 0.

Q1 : Proposer une détermination ascendante de la distance d'édition entre « chien » et « niche ».