- Informatique Tronc Commun : DS 2 -

16/11/23

I Plus longue sous-suite commune

A Présentation

On se donne deux mots, c'est à dire deux chaines de caractères `A` et `B`.

Une sous-suite commune de `A` et `B` est une chaine de caractères telle qu'on la retrouve parmi les lettres de `A` et `B`, les trois étant parcourues dans le même ordre.

1er exemple : lorsque `A='abcdef'` et `B='agbcjkf'`, voici des sous-suites communes de A et B :
`'a'` `'b'` `'bf'` `'abcf'` ...
parmi elles, la plus longue est `'abcf'`

2e exemple : lorsque `A='Le chat mange une souris rebelle'`
et `B='Ces chats blancs chassent une hirondelle'`,
la plus longue sous suite commune entre `A` et `B` est `'e chat ane une irelle'`

Parmi les différentes applications à cette recherche de plus longue sous-suite commune, on peut citer :

- Biologie : L'ADN est décrit par une suite de lettres formée à partir des lettres ATCG. Il arrive qu'il y ait des insertions dans l'ADN. On peut alors comparer deux informations génétiques et identifier qu'un ADN source a été complété par des lettres via des modifications génétiques, par exemple, AGTCGTCGATCGTAA devenant AGATCGTCGAGTCGTAA. L'algorithme identifiera dans le second la sous-suite du premier.
- Informatique : mise en place d' algorithmes d'anti-plagia en identifiant une longue phrase commune entre deux fichiers
- Correction orthographique : Si on tape par erreur «infomatique» au lieu de «informatique», un algorithme peux identifier automatiquement que le mot attendu était «informatique» car ces deux mots ont la même plus longue sous-suite commune «infomatique»

On revient au cas général :

On se donne deux chaines de caractères $A = a_0a_1...a_{n-1}$ et $B = b_0b_1...b_{p-1}$, de tailles respectives n et p.

On appelle plus longue sous suite commune (PLSSC) toute suite de A qui est aussi une sous suite de B et qui est de longueur maximale. Chercher une telle sous-suite revient à chercher deux suites strictement croissantes $(i(k))_{0\leqslant k\leqslant L-1}$ et $(j(k))_{0\leqslant k\leqslant L-1}$ et L maximal tels que $a_{i(k)} = b_{j(k)}$ pour tout $k \in \{0, ..., L-1\}$.

Q 1 Les sous-suites de A correspondent aux parties de l'ensemble des lettres qui constituent A ; il y en a donc autant que le nombre de parties d'un ensemble à n éléments.

Il y a donc 2^n sous-suites de A.

Un algorithme basé sur une recherche exhaustive devrait comparer chaque sous-suite de A avec chaque sous-suite de B, ce qui correspond à $2^n \times 2^p$ comparaisons. L'algorithme aurait alors une complexité au moins exponentielle ce qui le discalifie.

B Programmation Dynamique

Nous allons voir que ce problème de recherche de PLSSC peut être résolu de façon bien plus efficace en faisant appel à la programmation dynamique. Pour cela, nous définissons le sous-problème suivant :

Pour $1 \leqslant i \leqslant n-1$ et $1 \leqslant j \leqslant p-1$, on note $c(i,j)$ la longueur de la plus longue sous suite commune entre les suites tronquées $a_0a_1...a_{i-1}$ (i premiers symboles de A, ce qui correspond à `A[:i]`) et $b_0b_1...b_{j-1}$ (j premiers symboles de B, ce qui correspond à `B[:j]`)

Q 2 On choisit de plus comme convention : $c(i,j) = 0$ si $i = 0$ ou $j = 0$.

Si $i = 0$, alors la suite tronquée de A que l'on considère est la suite vide qui a une seule sous-suite : elle même. C'est donc la seule sous-suite commune de $A[:i]$ et $B[:i]$ et elle est de longueur 0, ce qui justifie que dans ce cas : $c(i,j) = 0$.

De même, si $j = 0$, la suite tronquée de B est vide et $c(i,j) = 0$.

Q 3 Soient $1 \leqslant i \leqslant n-1$ et $1 \leqslant j \leqslant p-1$. La fonction c vérifie :

$$c(i,j) = \begin{cases} 1 + c(i-1, j-1) & \text{si } a_{i-1} = b_{j-1} \\ max\big(c(i-1,j), c(i,j-1)\big) & \text{si } a_{i-1} \neq b_{j-1} \end{cases}$$

On suppose $a_{i-1} = b_{j-1}$. Nous allons démontrer l'égalité par double inégalité.

- On pose $k = c(i-1, j-1)$, il existe alors une sous-suite $x_0 \dots x_{k-1}$ commune à $A[:i-1]$ et $B[:j-1]$. Donc : $x_0 \dots x_{k-1}a_i$ est commune à $A[:i]$ et $B[:j]$ et a pour longueur $k+1 = 1 + c(i-1, j-1)$. Donc la plus longue sous-suite commune de $A[:i]$ et $B[:j]$ est de longueur au moins $1 + c(i-1, j-1)$, i.e. : $c(i,j) \geq 1 + c(i-1, j-1)$.
- On pose $m = c(i,j)$ et $x_0 \dots x_{m-1}$ une sous-suite commune à $A[:i]$ et $B[:j]$, comme $a_i = b_j$, $m \geq 1$ car la sous-liste a_{i-1} est commune à $A[:i]$ et $B[:j]$. Donc $x_0 \dots x_{m-2}$ est une sous-suite commune à $A[:i-1]$ et $B[:j-1]$ et elle est de longueur $m-1$, la plus longue sous-suite commune à $A[i-1]$ et $B[j-1]$ est donc de longueur au moins $m-1$, d'où : $c(i-1, j-1) \geq c(i,j) - 1$.

Donc par double inégalité, $c(i,j) = 1 + c(i-1, j-1)$.

remarque : la seconde inégalité se démontre de la même manière, mais on distingue deux cas pour la seconde inégalité suivant que $x_{m-1} = a_{i-1}$ ou $x_{m-1} \neq a_{i-1}$.

Q 4 $c(n,p)$ donne la plus longue sous-suite commune à A et B car on obtient alors la longueur de la PLSSC de $A[:n] = A$ et $B[:p] = B$.

Q 5 `memo_longueur` dans le fichier `ds1_corrige.py`.

Q 6 `bottom_up_longueur`

Q 7 On compte comme opérations élémentaires les affectations, les opérations algébriques sur les nombres, leur comparaison, la lecture ou écriture dans une liste, un dictionnaire, l'appel à la fonction `len` et `max` pour 2 arguments numériques.

- Avant les boucles, il y a 3 opérations élémentaires.
- Pour la première boucle sur `i`, il y a $n+1$ tours avec une opération à chaque tour, donc $n+1$ opérations en tout.
- De même pour la boucle sur j : $p+1$ opérations.
- Chaque tour des boucles imbriquées a moins de 10 opérations et il y a $n \times p$ tours ; ce qui donne au plus $10np$ opérations.

Le nombre d'opérations est inférieur à : $10np + n + p + 5 = O(np)$.

La complexité de `bottom_up_longueur` est donc en $O(np)$.

Q 8 `PLSSC`

II Points proches dans le plan

Ce problème, pouvant par exemple survenir dans le domaine de la navigation maritime, vise à déterminer, dans un nuage de points du plan, la paire de points les plus proches. Il est constitué de trois parties dépendantes.

Formellement, on suppose qu'on dispose de $n(\geqslant 2)$ points dans le plan $(M_0, M_1, \ldots, M_{n-1})$ dans un ordre quelconque pour le moment. Ils seront représentés en Python par deux listes de flottants de taille n : `coords_x` et `coords_y`, donnant respectivement les abscisses et les ordonnées des points.

On dira ainsi que M_i est le point d'indice i, qu'il a pour abscisse x_i =`coords_x`[i] et pour ordonnée y_i =`coords_y`[i].

On supposera que `coords_x` et `coords_y` sont des variables globales, qu'on ne modifiera jamais au cours de l'exécution de l'algorithme.

A Approche exhaustive

On utilise la distance euclidienne définie par $d(M_i, M_j) = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$.

Q 1 `distance(`i, j`)`

Q 2 `plus_proche()`

Q 3 Avec les mêmes opérations élémentaires que dans la partie précédente. On remarque que un appel à `distance` coute 6 opérations élémentaires.

- Avant les boucles, il y a 12 opérations élémentaires.
- Un tour de la boucle double représente au plus 12 opérations. Pour chaque i de 0 à n exclu il y a $n - (i + 1)$ tours. Le nombre d'opérations est alors inférieur à :

$$\sum_{i=0}^{n-1}(n - i - 1) \times 12 = O(n^2).$$

La complexité est donc quadratique en n : $O(n^2)$

B Quelques outils pour s'améliorer

On souhaite maintenant obtenir la distance entre les deux points les plus proches avec une meilleure complexité. Pour cela nous allons décrire un algorithme utilisant une méthode de type diviser pour régner. Cette partie introduit des fonctions utiles pour la mise en œuvre de cet algorithme.

On se donne la fonction suivante :

```
1 def tri(liste):
2     n=len(liste)
3     for i in range(n):
4         pos=i
5         while pos>0 and liste[pos]<liste[pos-1]:
6             liste[pos],liste[pos-1]=liste[pos-1],liste[pos]
7             pos = pos-1
```

Q 4 Cette fonction renvoie *None* (pas de `return`). Elle modifie la liste passée en argument pour la trier dans l'ordre croissant.

Un invariant de boucle : $\mathcal{H}(k)$: « avant le tour pour $i = k$, la liste `liste[:k]` est triée »pour $k \in \{0, 1, \ldots, n\}$ avec $\mathcal{H}(n)$: « la liste est triée après le tour pour $i = n - 1$ ».

Q 5

- Avant la boucle : 2 opérations.
- La boucle while fait au plus i tours et 12 opérations par tour ; et cela pour chaque i de 0 à n exlus par la boucle for. Ce qui fait au plus :

$$\sum_{i=0}^{n-1}(1 + 12i) = n + 6 \times n(n - 1)$$

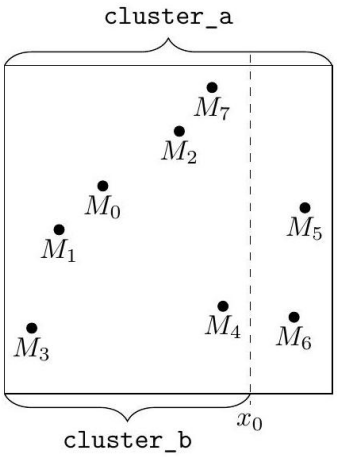
La complexité est donc quadratique : $O(n^2)$.

Q 6 `tri2`

Q 7 L'algorithme de tri fusion est de complexité quasi-linéaire : $O(nlg(n))$ dans le pire des cas.
remarque : le tri rapide est de complexité quasi-linéaire en moyenne, mais quadratique dans le pire des cas.

On admettra que l'on dispose de deux listes de n entiers `liste_x` (resp. `liste_y`) contenant les indices des points du nuage triés par abscisses croissantes (resp. par ordonnées croissantes). On supposera désormais que deux points quelconques ont des abscisses et des ordonnées distinctes.

Dans toute la suite, un sous-ensemble de points sera décrit par un cluster. Un cluster est une matrice de deux lignes contenant chacune les mêmes numéros correspondant aux numéros des points dans le sous-ensemble considéré. Dans la première ligne, les points sont triés par abscisses croissantes ; dans la seconde, ils sont triés par ordonnées croissantes. La figure 1 donne la représentation de deux clusters.




```
liste_x = [3, 1, 0, 2, 7, 4, 6, 5]
liste_y = [3, 6, 4, 1, 5, 0, 2, 7]

cluster_a = [liste_x, liste_y]

cluster_b = [ [3, 1, 0, 2, 7, 4],
              [3, 4, 1, 0, 2, 7] ]
```

Figure 1 - Représentation en Python de deux clusters

Pour être efficace, notre algorithme ne doit pas re-trier les listes des indices de points à chaque étape. Nous allons donc définir une fonction qui permet d'extraire des indices d'un cluster et former ainsi un nouveau cluster plus petit.

Q 8 `sous_cluster (cl, x_min, x_max)`

Q 9 `mediane(cl)`

C Méthode sophistiquée

Le fonctionnement de l'algorithme utilisant une méthode de type diviser pour régner est illustré par la fig. 2 :

1. Si le cluster contient deux ou trois points, on calcule la distance minimale en calculant toutes les distances possibles.

2. Sinon, on sépare le cluster en deux parties G et D qu'on supposera de tailles égales (éventuellement à un point près) suivant la médiane des abscisses, qu'on notera x_0.

3. Les deux points les plus proches sont soit tous les deux dans G, soit tous les deux dans D, soit un dans G et un dans D.

4. On calcule récursivement le couple le plus proche dans G et le couple le plus proche dans D. On note d_0 la plus petite des deux distances obtenues.

5. On cherche s'il existe une paire de points (M_1, M_2) telle que M_1 est dans G, M_2 dans D, et $d(M_1, M_2) < d_0$.

6. Si on en trouve une (ou plusieurs), on renvoie la plus petite de ces distances. Sinon, on renvoie d_0.

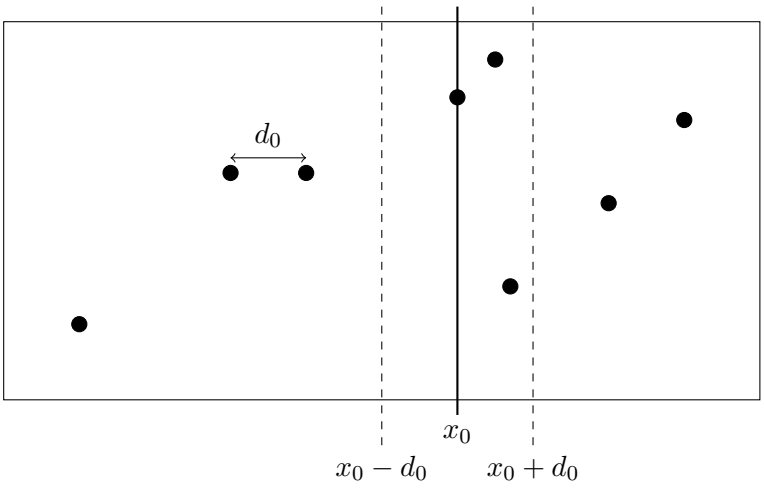


figure 2 - Illustration du diviser pour régner

Q 10 `gauche(cl)`

On suppose qu'on dispose d'une fonction `droite(cl)` qui renvoie le cluster contenant tous les autres points du cluster cl n'appartenant pas au cluster renvoyé par la fonction `gauche(cl)`.

Q 11 Soit $M_1 \in G$ et $M_2 \in D$ et l'abscisse de M_1 est strictement inférieure à $x_0 - d_0$, alors d'après l'inégalité de Pythagore donne que la distance entre M_1 et M_2 est supérieur à $x_{M_2} - x_{M_1}$ (la différence des abscisses) qui est alors strictement supérieure à d_0, donc il n'est pas nécessaire de considérer M_1 et M_2 dans la recherche de deux points les plus proches.

De même si $M_1 \in G$ et l'abscisse de M_2 est strictement supérieure à $x_0 + d_0$, alors la distance entre M_1 et M_2 est strictement supérieure à d_0.

Ainsi on peut se contenter de chercher les points M_1 et M_2 de l'étape 5 de l'algorithme dans l'ensemble des points dont l'abscisse appartient à $I_0 = [x_0 - d_0, x_0 + d_0]$.

Q 12 `bande_centrale(cl,d0)`

Q 13 **On admet le résultat de cette question**

Montrer que deux points M_1 et M_2 (de l'étape 5 de l'algorithme) situés à une distance inférieure à d_0 se trouvent, dans la deuxième ligne du cluster (c'est-à-dire la ligne triée par ordonnées croissantes), séparés d'au plus 6 éléments.

On pourra montrer par l'absurde qu'un rectangle, à préciser, de dimensions $2d_0 \times d_0$ contient au plus 8 points.

Q 14 `fusion(cl,d0)`

La fonction `bande_centrale` est de complexité linéaire en la taille n du cluster, chaque tour de boucle coute au plus 12 opérations (appel à `distance` comprise) et il y a au plus 8 tour pour j et n tours pour i, donc au plus $8 \times 12 \times n$ opérations dans les boucles.

La complexité de la fonction est donc linéaire en la taille du cluster.

Q 15 `distance_minimale(cl)`