Devoir Bilan d'Informatique : corrigé

PTSI Lycée Eiffel

4 juin 2021

I. Génération d'une séquence aléatoire d'ADN.

1. (a) La commande renvoie le cinquième caractère de la chaîne, donc 'T'.
   (b) La commande renvoie la tranche comprise entre les positions 2 (incluse) et 6 (non incluse), donc 'TGTA'.
   (c) La commande renvoie le nombre de caractères de la chaîne, donc 11.
   (d) On va afficher la suite des caractères de la chaîne, en ordre alphabétique inverse, donc 'TTTGGGCCAAA'.
2. On essaie de bien faire attention à renvoyer une chaîne de caractères et pas une liste, et on n'oublie pas d'importer le module random. Par exemple :

```
from random import randint
def seqaleatoire(n) :
    s=''
    for i in rang(n) :
        r=randint(1,4)
        if r==1 :
            s=s+'A'
        elif r==2 :
            s=s+'C'
        elif r==3 :
            s=s+'G'
        else :
            s=s+'T'
    return s
```

3. Il est bien entendu interdit d'utiliser directement un count, le programme est facile à écrire à la main :

```
def compteA(s) :
    c=0
    for i in s :
        if i=='A' :
            c=c+1
    return c
```

4. On peut utiliser des programmes semblables à celui écrit pour la question précédente, mais si on veut un programme vaguement optimisé, il est préférable de ne faire qu'une seule boucle au lieu de parcourir quatre fois la chaîne de caractères :

```
def comptenucleo(s) :
    l=[0,0,0,0]
    for i in s :
        if i=='A' :
            l[0]=l[0]+1
        elif i=='C' :
            l[1]=l[1]+1
        elif i=='G' :
            l[2]=l[2]+1
        else :
            l[3]=l[3]+1
    return l
```

5. On a donc environ 3×10^9 nucléotides à stocker. Si ce stockage est vraiment optimisé, on doit pouvoir faire tenir un nucléotide sur 2 bits (quatre informations différentes, par exemple on décide que 00 correspond au 'A', 01 au 'C', 10 au 'G' et 11 au 'T'), donc quatrenucléotides par octet. On aurait alors besoin de $\frac{3 \times 10^9}{4}$ octets, soit 750 Mo. Théoriquement, la clé USB est suffisante.

II. Recherche d'un motif par algorithme naïf.

1. Il ne resterait plus assez de lettres pour avoir le motif entier si on répère le début du motif à un rang supérieur à $len(s) - len(m)$.
2. On parcourt simplement toute la liste s (ou presque), et à chaque fois, on teste sur les caractères suivants si on a repéré le motif. Bien sûr, si on repère le motif entier, on arrête la fonction via un return.

```
def recherchermotif(s,m) :
    n=len(m)
    for i in range(len(s)-n) :
        a=0
        while a<n and s[i+a]==m[a] :
            a=a+1
        if a==n :
            return i
    return -1
```

3. On effectue au pire (si on ne trouve jamais le motif) $len(s) - n$ passages dans la boucle principale, et chaque passage dans la boucle nécessite au maximum n comparaisons, si on a systématiquement les $n - 1$ premières lettres correctes mais que la dernière ne correspond pas à celle du motif (on notera qu'il est difficile que ça se produise vraiment pour toutes les valeurs de i mais l'estimation globale reste correcte). Au total, on aura donc au maximum $n \times (len(m) - n)$ comparaisons, en ayant noté n la longueur du motif recherché.
4. On effectuera donc au maximum $50 \times (3 \times 10^9 - 50)$ comparaisons, donc à peu près 150 milliards de comparaisons. Avec 10^{12} comparaisons par seconde, on aura besoin d'un temps de recherche de 0.15 seconde.

5. On va donc avoir $\frac{10^9}{50} = 2 \times 10^7$ morceaux à comparer au deuxième génome, ce qui prendra à chaque fois 0.05 seconde (lé génome étant trois fois moins long qu'à la question précédente, le temps sera trois fois moins grand). On devrait donc attendre quand même 10^6 secondes. Un million de secondes, c'est grosso modo 250 heures (on divise par 3 600), donc plus de dix jours. On peut considérer que l'algorithme n'est pas vraiment utilisable en pratique.

III. Recherche d'un motif par l'algorithme de Knuth-Morris-Pratt.

1. Une question cadeau. Les préfixes sont 'A', 'AT', 'ATG', 'ATGC', 'ATGCG' et 'ATGCGT'. Les suffixes sont 'A', 'TA', 'GTA', 'CGTA', 'GCGTA' et 'TGCGTA'.
2. Dans le premier cas c'est 'AC', dans le deuxième cas 'TAC'.
3. (a) La fonction renvoie une liste de nombre entiers.

   (b) Il y a quatre erreurs manifestes :
   - la variable n n'est jamais définie, elle devrait être remplacée par $len(m)$
   - le simple égal doit être remplacé par un == dans le test du premier if
   - il manque un : derrière le premier else
   - le return final est clairement mal indentée (sinon la fonction est stupide !), il devrait être à hauteur des premières affectations et du while

   (c) On va avoir les étapes suivantes :
   - $1 < 6$, donc on effectue un premier passage dans la boucle. Comme $m[0] \neq m[1]$ et $j = 0$, on ajoute un 0 dans la liste L et on incrémente la variable i. On a donc après ce premier passage $i = 2$, $j = 0$ et $L = [0, 0]$.
   - $2 < 6$, deuxième passage dans la boucle. Cette fois $m[2] = m[0]$, donc on augmente i et j et on ajoute un 1 dans la liste. Après ce passage, $i = 3$, $j = 1$ et $L = [0, 0, 1]$.
   - $3 < 6$, troisième passage dans la boucle. Cette fois $m[3] \neq m[1]$ et $j > 0$, la seule modification est la réinitialisation de j, et on aura $i = 3$, $j = 0$ et $L = [0, 0, 1]$.
   - on a toujours $3 < 6$, mais $m[3] = m[0]$, donc on augmente i et j, et on ajoute 1 à la liste, pour obtenir $i = 4$, $j = 1$ et $L = [0, 0, 1, 1]$.
   - $4 < 6$ nouveau passage dans la boucle, et $m[4] = m[1]$, modifications similaires à l'étape précédente, $i = 5$, $j = 2$ et $L = [0, 0, 1, 1, 2]$.
   - $5 < 6$ et encore une fois $m[5] = m[2]$, on continue sur notre lancée : $i = 6$, $j = 3$ et $L = [0, 0, 1, 1, 2, 3]$.
   - il est temps de s'arrêter, le programme renvoie $[0, 0, 1, 1, 2, 3]$.
4. (a) Elles représentent simplements des indices de parcours des chaînes de caractères s (pour la variable i) et m (pour j).

   (b) Si on atteint la condition $j = len(m) - 1$ dans la boule while, cela signifie qu'on a trouvé le motif dans notre chaîne de caractères (on a repéré le dernier caractères, et tous les précédentes ont déjà été testés auparavant). Il est donc temps d'arrêter la rechercher et de renvoyer l'indice $i - j$ qui correspond bien à l'indice de début d'apparition du motif.

   (c) La liste L sert à éviter les tests inutiles en « sautant » de temps à autre des caractères dans le parcours de la chaîne s. Reprenons l'exemple étudié juste avant, où le motif recherché est 'ACAACA'. Imaginons qu'en cours de parcours de s, on repère le début du motif, mais que la coïncide ne fonctionne plus au cinquième caractère. Cela signifie donc qu'on a dans notre chaîne un morceau du genre 'ACAAX' avec un 'X' qui n'est pas un 'C' (sinon on aurait continué). Dans ce cas, il est complètement inutile de chercher le motif à partir du 'C' (qui ne correspond pas à la première lettre du motif), mais aussi à partir du 'A' qui suit le 'C' (ce 'A' est bien la première lettre du motif, mais il est suivi d'un autre 'A' qui ne

peut pas être correct). Par contre, on peut tester la présence du motif à partir du dernier 'A' (si le 'X' qui suit est un 'C' et que la suite correspond, on aura trouvé le motif ; en fait on sait déjà que ce 'X' n'est pas un 'C' mais l'algorithme ne gère pas cela). On peut donc « sauter » deux caractères dans le parcours de 's', c'est à ça que correspond le 2 qui était en cinquième position de la liste L.

IV. Recherche d'un motif en utilisant une structure de liste.

1. Il n'y aucune raison ici de ne pas considérer la chaîne toute entière comme une sous-chaîne. On a donc les sous-chaînes suivantes : 'A', 'C', 'T', 'G', 'AC', 'CT', 'TC', 'CG', 'ACT', 'CTC', 'TCG', 'ACTC', 'CTCG', 'ACTCG'.
2. Au maximum, on aura n sous-chaînes à un seul caractère (ici, en pratique, on sait bien que le nombre de sous-chaînes distinctes à un seul caractère ne peut pas dépasser quatre, mais on n'en tiendra pas compte pour l'estimation), $n-1$ à deux caractères, etc, jusqu'à une seule chaîne à n caractères. Au total donc : $n+(n-1)+\cdots+1 = \dfrac{n(n+1)}{2}$ sous-chaînes distinctes.
3. Le fait qu'il s'agisse de chaînes de caractères n'a pas d'incidence sur le programme puisqu'on peut les comparer alphabétiquement à l'aide de $<$:

```
def indicemin(L) :
    i,m=0,L[0]
    for j in range(1,len(L)) :
        if L[j]<m :
            i,m=j,L[j]
    return i
```

4. (a) L'opération + effectue une concaténation, autrement dit une « mise bout à bout ».
   (b) C'est un programme récursif puisque le programme lui-même est appelé dans le return.
   (c) Il trie la liste par ordre croissant. En effet, il repère (en utilisant la fonction indicemin) le plus petit élément de la liste, le positionne en tête de liste (en l'éliminant de la position où il était initialement) et recommence sur la liste restante.
5. Si on a d'abord trié la liste par ordre alphabétique, c'est évidemment pour en tirer profit ! Un programme du type balayage de toute la liste serait donc complètement inadapté, puisqu'il est très lent et qu'il fonctionnerait tout aussi bien avec une liste non triée. On privilégierait ici une approche dichotomique, beaucoup plus rapide.

V. Exploitation d'un base de données.

1. La requête va afficher le nombre de séquençages effectués le 5 juin 2012.
2. SELECT nadn FROM sequencage WHERE gene='leuS'
3. SELECT employe FROM sequencage WHERE employe LIKE 'G%'
   Notons que cette requête risque bien sûr d'afficher beaucoup de fois chaque nom commençant par G, si on veut un affichage unique de chaque nom, on ajoute un DISTINCT derrière le SELECT, mais nous n'en avions pas parlé en cours.
4. SELECT gene FROM sequencage JOIN adn ON sequencage.nadn=adn.adn WHERE genre='Bacillus'
   Ici, comme l'attribut commun aux deux tables porte le même nom dans les deux tables, on peut alléger la syntaxe en appliquant un NATURAL JOIN qui permet de ne pas mettre du tout de ON ensuite.

5. SELECT genre, espece FROM adn JOIN sequencage ON adn.nadn=sequencage.nadn WHERE employe='Dupont' AND date='10-03-2018'

   On a conservé le format proposé pour les dates, qui n'est toutefois pas le format standard en SQL.