Correction DM - Autour du séquençage du génome

January 22, 2025

0.1 Correction DM - Autour du séquençage du génome

`seq='ATCGTACGTACG'`

Q1. Que renvoie la commande `seq[3]` ? Que renvoie la commande `seq[2:6]` ?

`seq[3]` est le 4^{me} élément de la liste donc `seq[3] = 'G'`

`seq[2:6]` est la tranche de liste allant de `seq[2]` à `seq[5]`, `seq[2:6] = 'CGTA'`

Q2. Écrire une fonction generation() qui prend en paramètre un entier n et qui renvoie une chaîne de caractères aléatoires de longueur n ne contenant que des 'A', 'C', 'G' et 'T'

Pour respecter le principe de génération, nous allons associer à chaque nombre entier aléatoire i entre 1 et 4, `lettre[i - 1]`avec `lettre = ['A','C','G','T']`

[1]:
```
from random import randint

def generation(n: int) -> str:
    lettre = ['A','C','G','T']
    sequence = [] # Initialisation de la séquence vide

    for _ in range(n):
        i = randint(1,4) # Les bornes de randint sont inclues
        l = lettre[i - 1]
        sequence.append(l)

    return "".join(sequence) # join permet de transformer un liste en châines␣
↪de caractères.
```

Q3. Que fait la fonction mystere(seq) qui prend en argument une séquence d'ADN 'seq' (une chaîne de caractères ne contenant que des 'A', 'C', 'G' et 'T') ?

La fonction mystère initialise des variables a,c,g et t à 0. Ainsi que i à `len(seq) - 1`.

Lors du premier tour de la boucle while, `seq[i]`vaut donc la dernière lettre de la séquence.

Les if/elif successifs permettent d'incrémenter la variable associée à la lettre obtenue.

`i`est décrementé de 1. Ce qui permet de parcourir la séquence dans l'ordre décroissant.

La boucle while continue jusqu'à la fin du traitement de toutes les lettres séquence `i >=0`.

Ainsi cette fonction compte les nombres d'occurences des lettre 'a', 'c', 'g', 't' dans la séquence.

Elle retourne une liste des taux d'apparitions de ces lettres en pourcentage.

Q4. Quelle est la complexité de la fonction mystere() ?

Donner le nom de la variable permettant de montrer la terminaison de l'algorithme.

La terminaison de la fonction est déterminée par la sortie de la boucle while. La variable `i` est initialisé avec une valeur positive égale à la longueur de la séqence - 1.

Toute suite strictement décroissante dans N atteint 0. Donc la boucle while se termine.

Le nombre de tour de boucle est donc égal à `n`. La complexité de l'intérieur de la boucle est en $\mathcal{O}(4) + \mathcal{O}(2) = \mathcal{O}(1)$.

Ainsi comme de nombreux algorithmes de parcours séquentiels, la compléxité de la fonction mystère est en $\mathcal{O}(n)$.

[4]:
```
def recherche(S: list, M: list) -> int:
    '''
    Recherche la sous-chaine M dans S.
    Retourne -1 si M n'est pas présente dans S.
    La position de la première lettre de M dans S à l'endroit où M et S␣
↪s'alignent.
    '''

    nb_trouvees = 0 # Nombre de lettres correspondantes entre M et une␣
↪sous-chaine de S.
    i = 0 # indice du début de comparaison de M et sous-chaine de S qui␣
↪commence à S[i]

    while i <=  len(S) - len(M): # La sous-chaine commence au plus tard à␣
↪l'indice len(S) - len(M)
        while M[nb_trouvees] == S[i + nb_trouvees]: # lettres correspondantes
            nb_trouvees += 1 # Pour tester la lettre suivante
            if nb_trouvees == len(M): # la sous_chaine a été trouvées
                return i

        i += 1 # Pour décaler les test de comparaisons dans S

    return -1 # Sous-chaîne M non trouvée dans S
```

Q6. Combien faut-il d'opérations pour chercher un motif de 50 caractères dans une séquence d'ADN en utilisant l'algorithme naïf ? On supposera qu'une séquence d'ADN est composée de 3.10^9 caractères. En combien de temps un ordinateur réalisant 10^{12} opérations par seconde fait-il ce calcul ?

La compléxité est en $\mathcal{O}(nm)$ ce qui indique que le nombre d'opérations atomiques nécessaires à son exécution est inférieur à Cnm avec $C \in R^+$.

Ici elle est égale à quelques unité près dans le pire des cas à nm.

Donc le temps en seconde est pour réaliser le calcul est estimé à:

Temps $= \frac{50.3.(3.10^9-49)}{10^{12}} \approx 0,15s$

Q7. En utilisant les calculs précédents, combien de temps faut-il pour un ordinateur réalisant

opérations par seconde pour comparer deux séquences d'ADN avec l'algorithme naïf ?

Vous semble-t-il intéressant d'utiliser l'algorithme de recherche naïf ?

Avec le résultat de la quesion précédente, si la deuxième séquence est de taille comparable à la première, elle sera composée de $3.10^9/50 \approx 6.10^7$ séquences de 50 caractères.

il faudra alors $0,15.6.10^7 = 9.10^6 s \approx 104,166 jours$

L'algorithme naif est long à s'éxécuter.

Q8. Donner tous les préfixes et les suffixes du motif 'ACGTAC'.

En augmentant la taille des préfixes et des suffixes et en ignorant les mots vides. Nous avons :

Les préfixes de 'ACGTAC' sont A,AC,ACG,ACGT,ACGTA.

Les suffixes de 'ACGTAC' sont C,AC,TAC,GTAC,CGTAC.

Q9. Quel est le plus grand préfixe de 'ACGTAC' qui soit aussi un suffixe ?

Quel est le plus grand préfixe de 'ACAACA' qui soit aussi un suffixe ?

AC est le seul préfixe de 'ACGTAC' qui est aussi un suffixe

Donc ACA est le plus grand suffixe de 'ACAACA' qui soit aussi un préfixe.

Q10. Quel est le type de la sortie de la fonction fonctionannexe() ?

La fonction annexe retourne la variable `F`qui est initialisée à `[0]`et à laquelle on ajoute des nombres entiers avec `append`.

la fonction fonctionannexe() retourne donc une liste d'entiers.

Q11. Une ou des erreurs de syntaxe s'est (se sont) glissée(s) dans la fonction fonctionannexe().

Identifier la ou les erreur(s) et corriger la fonction pour qu'il n'y ait plus de message d'erreur quand on exécute la fonction.

Voici une version corrigée.

[15]:
```
def fonctionannexe(M):
        F=[0]
        i=1
        j=0
        m = len(M) # m n'était pas initialisé
        while i < m :
            if M[i]==M[j]: # le test d'agalité est avec == et non =
                F.append(j+1)
                i=i+1
```

```
            j=j+1
        elif j>0: # Nous utilisons elif, sinon nous aurions deux else
                j=F[j - 1]
        else:
            F.append(0)
            i=i+1
    return F
```

Q12. Décrire l'exécution de la fonction fonctionannexe() lorsque M='ACAACA'

en précisant sur le pour les six premiers tours dans la boucle while, à la sortie de la boucle, le contenu des variables : i, j et F.

Variables initiales

- **M** = 'ACAACA'
- **F** = [0] (tableau initialisé avec un premier élément 0)
- **i** = 1 (index pour parcourir M)
- **j** = 0 (index pour suivre le préfixe courant)
- **m** = 6 (longueur de M)

Étape par étape pour les 6 premiers tours de la boucle `while` :

Tour	**i**	**j**	**M[i]**	**M[j]**	Remarques	**F**
1	1	0	C	A	`C != A`, j reste 0, ajoute 0	[0, 0]
2	2	0	A	A	`A == A`, ajoute `j+1` (=1)	[0, 0, 1]
3	3	1	A	C	`A != C`, j != 0, j devient `F[j-1]` (=0), **pas d'ajout à F**	[0, 0, 1]
4	3	0	A	A	`A == A`, ajoute 1 à `j` (=1)	[0, 0, 1, 1]
5	4	1	C	C	`C == C`, ajoute 1 `j` (=2)	[0, 0, 1, 1, 2]
6	5	2	A	A	`A == C`, ajoute 1 à`j` (=3)	[0, 0, 1, 1, 2, 3]
6	6	3	-	-	Sortie de boucle i = m	[0, 0, 1, 1, 2, 3]

- **F** = [0, 0, 1, 1, 2, 3] (tableau des préfixes)

Le tableau des préfixes pour M = 'ACAACA' est : **[0, 0, 1, 1, 2, 3]**].

Q13. L'algorithme KMP.

Expliquer la ligne 2, expliquer les lignes 3 et 4.

Quelles lignes correspondent au cas où on a trouvé le mot ?

Que fait le programme dans ce cas ?

La ligne 2 calucle la fonction annexe pour, à l'aide des plus long suffixes dans les sous-chaine de M, éviter des comparaisons inutiles.

Les lignes 3 et 4 initialisent les variables `i` et `j` à 0 pour comparer les lettres de T et M.

Contrairement à l'algorithme naif, j pourra être modifié en utilisant le résultat de la fonction annexe.

Les liges 6 à 8 sont celles qui correspondent au cas où le motif a été trouvé :

Ligne 6 : les dernières lettres correspondent ligne 7 : nous vérifons que nous avons la taille de M ligne 8 retour de l'indice i'correspondant à la première lettre de M dans T à l'endroit où le motif est trouvé.

Q14.

[23]:
```
def triinsertion(liste):
    """
    Trie une liste de nombres en utilisant l'algorithme de tri par insertion.

    Paramètres :
        liste (list) : La liste de nombres à trier.

    Retourne :
        list : La liste triée.
    """
    for i in range(1, len(liste)):
        # L'élément à insérer
        valeur_insertion = liste[i]

        # Position où placer l'élément
        j = i - 1

        # Déplacer les éléments de la liste qui sont plus grands que␣
↪valeur_insertion
        while j >= 0 and liste[j] > valeur_insertion:
            liste[j + 1] = liste[j]
            j -= 1

        # Placer valeur_insertion à sa position correcte
        liste[j + 1] = valeur_insertion

    return liste
```

Q15. Comment peut-on adapter la fonction triinsertion() à une liste de chaîne de caractères ?

Pour adapter la fonction triinsertion() à une liste de chaînes de caractères, il faut comprarer la chaine qui se trouve avant l'autre.

En Pyhton, aucune modification structurelle n'est nécessaire.

Les comparaisons (>, <) fonctionnent aussi sur les chaînes en se basant sur leur ordre lexicographique (celui du dictionnaire).

Q16. Écrire une fonction recherchedichotomique() de recherche dichotomique dans une liste de nombres triés.

Voici ci-dessous le code de la fonction recherchedichotomique().

Son intérêt est de répondre à la quesion de la présence d'un élément dans une liste en $\mathcal{O}(\log(n))$ alors que la recherche séquentielle à une compléxité en $\mathcal{O}(n)$ dans le pire des cas.

Cependant cela demande que la liste soit triée.

[25]:
```python
def recherchedichotomique(liste, element):
    """
    Recherche un élément dans une liste triée en utilisant l'algorithme de␣
 ↪recherche dichotomique.

    Paramètres :
        liste (list) : La liste triée dans laquelle effectuer la recherche.
        element : L'élément à rechercher.

    Retourne :
        int : L'indice de l'élément s'il est trouvé, sinon -1.
    """
    debut = 0
    fin = len(liste) - 1

    while debut <= fin:
        milieu = (debut + fin) // 2

        # Vérifier si l'élément est au milieu
        if liste[milieu] == element:
            return milieu

        # Si l'élément est plus petit, il est dans la partie gauche
        elif liste[milieu] > element:
            fin = milieu - 1

        # Si l'élément est plus grand, il est dans la partie droite
        else:
            debut = milieu + 1

    # Si l'élément n'est pas trouvé
    return -1
```