DS1 /61

- **Important** : vous pouvez utiliser les fonctions des questions précédentes, même si vous ne les avez pas toutes démontrées.
- Vous devez répondre directement sur le Document Réponse (**DR**), soit à l'emplacement prévu pour la réponse lorsque celle-ci implique une rédaction, soit en complétant les différents programmes en langage Python.

Exercice : Cours /10

1. Écrire une fonction **Python** calculant la première position du maximum d'une liste `L`.

- **3 pts**

```python
def ind_max(L) :
    '' ind_max(L : list) -> i_max : int''
    n = len(L)
    i_max = 0
    for k in range(n) :
        if L[i] > L[i_max] :
            i_max = i
    return i_max
```

2. On se propose de coder quelques fonctions essentielles à la mise en place de l'algorithme du pivot de Gauss.

a) Programmer une fonction `zeroColSousDiag` qui prend en paramètre un numéro de colonne `j` et deux matrices numpy `M` et `N`, et qui renvoie le couple de matrices numpy `[G, D]` où :

- la variable `G` est modifiée afin d'obtenir, par une succession d'opérations élémentaires, une matrice dont les coefficients sous-diagonaux de la $j^{\text{ème}}$ colonne sont tous nuls,
- la variable `D` est modifiée par applications successives des mêmes opérations élémentaires que sur `G`.

- **3 pts**

```python
import numpy as np

def zeroColSousDiag(j, M, N) :
    '' zeroColSousDiag(j : int, M : np.matrix, N : np.matrix)''
        ''-> [G : np.matrix, D : np.matrix]''
    G = np.copy(M)
    D = np.copy(N)
    n = M.shape[0]
    for i in range(j+1, n) :
        D[i,:] = G[j,j] * D[i,:] - G[i,j] * D[j,:]
        G[i,:] = G[j,j] * G[i,:] - G[i,j] * G[j,:]
    return [G, D]
```

b) Quelle est la complexité de la fonction `zeroColSousDiag` ? Justifier.

- **2 pts :** $\underset{n\to+\infty}{\Theta}(n)$ **avec explications**

c) Programmer une fonction `trouvePivot` qui prend en paramètre un numéro de colonne `j` et une matrice numpy `M`, qui renvoie en sortie l'entier `p` calculant la ligne du premier coefficient sous-diagonal non nul de la $\texttt{j}^{\text{ème}}$ colonne de `M`. S'il n'y a pas de tel coefficient, la variable `p` devra contenir n, où n est le nombre de lignes de la matrice.

- **2 pts**

```
def trouvePivot(j, M) :
    ''trouvePivot(j : int, M : np.matrix) -> p : int''
    n = M.shape[0]
    for p in range(j, n) :
        if M[p, j] != 0 :
            return p
    return n
```

Problème /51

Dans ce sujet, on s'intéresse à la recherche d'un motif dans une molécule d'ADN.
Une molécule d'ADN est constituée de deux brins complémentaires, qui sont un long enchaînement de nucléotides de quatre types différents désignés par les lettres A, T, C et G. Les deux brins sont complémentaires : « en face » d'un 'A', il y a toujours un 'T' et « en face » d'un 'C', il y a toujours un 'G'. Pour simplifier le sujet, on va considérer qu'une molécule d'ADN est une chaîne de caractères sur l'alphabet {A,C,G,T} (on s'intéresse donc seulement à un des deux brins). On parlera de séquence d'ADN.

Partie I - Génération d'une séquence d'ADN

On considère la chaîne de caractère `seq = ATCGTACGTACG`.

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

- **1 pt : La commande `seq[3]` renvoie donc `'G'`**
- **1 pt : La commande `seq[2:6]` renvoie donc la chaîne de caractères `'CGTA'`**

4. Écrire une fonction `génération` 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'.
On pourra utiliser les fonctions `random` ou `randint` détaillées en **annexe**.

- **2 pts**

```
import numpy.random as rd

def generation(n) :
    seq = ''
    codage = {1 : 'A', 2 : 'C', 3 : 'G', 4 : 'T'}
    for k in range(n) :
        seq = seq + codage[rd.randint(1,5)]
    return seq
```

5. Que fait la fonction `mystere` qui prend en argument une séquence d'ADN `seq` (une chaîne de caractères ne contenant que des 'A', 'C', 'G' et 'T') ?
Le code de la fonction `mystere` se trouve dans le **DR 4**.

- **2 pts : La fonction `mystere` prend en argument une chaîne de caractère `seq` et renvoie la liste des pourcentages d'apparition des caractères `'A'`, `'C'`, `'G'` et `'T'` dans la chaîne `seq`, dans cet ordre.**

6. Quelle est la complexité de la fonction `mystere` ?
Donner le nom de la variable permettant de montrer la terminaison de l'algorithme (on justifiera le raisonnement).

- **1 pt : La complexité de la fonction `mystere` est en** $\underset{n\to+\infty}{\Theta}(n)$
- **2 pts : La variable permettant de démontrer la terminaison de l'algorithme est la variable `i` car elle définit une suite strictement décroissante d'entiers positifs. En effet, la variable `i` est initialisée à une valeur entière $n-1$ et décroît de 1 à chaque tour de boucle. C'est donc un variant de la boucle `while`. Cette dernière se termine donc.**

Partie II - Recherche d'un motif

II.1 - Algorithme naïf

Principe de l'algorithme naïf
On parcourt la chaîne. À chaque étape, on regarde si on a trouvé le bon motif. Si ce n'est pas le cas, on recommence avec l'élément suivant de la chaîne de caractères.

Cet algorithme a une complexité en $O(nm)$ avec n, la taille de la chaîne de caractère et m, la taille du motif.

7. Écrire une fonction `recherche` qui à une sous-chaîne de caractères `M` et une chaîne de caractères `S` renvoie -1 si `M` n'est pas dans `S` , et la position de la première lettre de la chaîne de caractères `M` si `M` est présente dans `S`.
Cet algorithme doit correspondre à l'algorithme naïf.

- **2 pts**

```
def recherche(M, S) :
    ''recherche(M : str, S : str) -> int''
    n = len(S)
    m = len(M)
    for k in range(n-m+1) :
        if S[k : k+m] == M :
            return k
    return -1
```

8. 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 $3\cdot10^9$ caractères. En combien de temps un ordinateur réalisant 10^{12} opérations par seconde fait-il ce calcul ?

- **1 pt : Pour un motif de $m = 50$ caractères dans une séquence d'ADN de $n = 3\cdot10^9$ caractères, il faut effectuer de l'ordre de $nm = 15\cdot10^{10}$ opérations.**
- **1 pt : Un ordinateur effectuera la recherche du motif en approximativement $0,15$ secondes**

En génétique, on utilise des algorithmes de recherche pour identifier les similarités entre deux séquences d'ADN. Pour cela, on procède de la manière suivante :

- découper la première séquence d'ADN en morceaux de taille 50 ;
- rechercher chaque morceau dans la deuxième séquence d'ADN.

9. En utilisant les calculs précédents, combien de temps faut-il pour un ordinateur réalisant 10^{12} 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 ?

- **1 pt : La comparaison de 2 séquences d'ADN nécessite donc 2500 heures donc plus de 3 mois. Ce temps étant déraisonnablement long, il n'est pas pertinent d'utiliser l'algorithme de recherche naïf.**

II.2 - Algorithme de Knuth-Morris-Pratt (1970)

II.2.a Préfixe et suffixe

Un préfixe d'un motif M est un motif u, différent de M, qui est un début de M.
Par exemple, 'mo' et 'm' sont des préfixes de 'mot', mais 'o' n'est pas un préfixe de 'mot'.

Un suffixe d'un motif M est un motif u, différent de M, qui est une fin de M.
Par exemple, 'ot' et 't' sont des suffixes de 'mot', mais 'mot' n'est pas un suffixe de 'mot'.

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

- **1 pt : Les préfixes du motif 'ACGTAC' sont : 'A', 'AC', 'ACG', 'ACGT' et 'ACGTA'.**
- **1 pt : Les suffixes du motif 'ACGTAC' sont : 'C', 'AC', 'TAC', 'GTAC' et 'CGTAC'.**

11. 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 ?

- **1 pt : Le plus grand préfixe de 'ACGTAC' qui soit aussi un suffixe est donc le motif 'AC'.**
- **1 pt : Le plus grand préfixe de 'ACAACA' qui soit aussi un suffixe est 'ACA'.**

II.2.b Algorithme de Knuth-Morris-Pratt

12. Quel est le type de la sortie de la fonction `fonctionannexe` ?

- **1 pt : Le type de la variable de sortie de la fonction `fonctionannexe` est une liste d'entiers (`list(int)`)**

13. 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 compile la fonction.

- **1 pt :**

```
2        m = len(M)
```

- **1 pt :**

```
6            if M[i] == M[j] :
```

14. Décrire l'exécution de la fonction `fonctionannexe` lorsque `M` $=$ `'ACAACA'` en précisant sur le **DR 7**, pour les six premiers tours dans la boucle while, à la sortie de la boucle, le contenu des variables : `i`, `j` et `F`.

- **3 pts : 1 pt toutes les 2 lignes**
  - × **à l'issue du premier tour de boucle `while` :** `i = 2, j = 0 et F = [0, 0]`
  - × **à l'issue du deuxième tour de boucle `while` :** `i = 3, j = 1 et F = [0, 0, 1]`
  - × **à l'issue du troisième tour de boucle `while` :** `i = 3, j = 0 et F = [0, 0, 1]`
  - × **à l'issue du quatrième tour de boucle `while` :** `i = 4, j = 1 et F = [0, 0, 1, 1]`
  - × **à l'issue du cinquième tour de boucle `while` :** `i = 5, j = 2 et F = [0, 0, 1, 1, 2]`
  - × **à l'issue du cinquième tour de boucle `while` :** `i = 6, j = 3 et F = [0, 0, 1, 1, 2, 3]`

15. Expliquer et commenter les groupements de lignes de l'algorithme KMP donnés dans le **DR 8**.

- **1 pt : la ligne 2 permet de stocker dans la variable `F` la liste où le `i`$^{\text{ème}}$ contient le nombre de caractères du plus grand préfixe de `M` qui est un suffixe de `M[: i+1]`**
- **1 pt : la ligne 3 initialise la variable `i` à 0 et la ligne 4 initialise la variable `j` à 0**
- **1 pt : les lignes correspondant au cas où on a trouvé le mot sont les lignes 7 et 8**
- **1 pt : dans le cas où on a trouvé le mot, la fonction renvoie l'indice de première apparition du mot `M` dans la chaîne de caractère `T`**
- **1 pt : les lignes correspondant au cas où on a trouvé deux lettres identiques sont les lignes 6 à 11**
- **1 pt : dans le cas où on a trouvé deux lettres identiques :**
  - × **soit on a trouvé le mot et on est ramené à la question précédente**
  - × **soit ce n'est pas le cas et on poursuit la recherche en passant au caractère suivant à la fois dans la chaîne de caractère `M` et la chaîne de caractère `T`**
- **1 pt : les lignes correspondant au cas où on a trouvé deux lettres différentes sont les lignes 12 à 16**
- **2 pts : dans le cas où on a trouvé deux lettres différentes :**
  - × **soit `j` $= 0$, ce qui signifie que l'on comparait `T[i]` au 1$^{\text{er}}$ caractère de `M`. Comme ces caractères sont différents, on poursuit la recherche en passant au caractère suivant dans `T`**
  - × **soit `j` > 0, ce qui signifie que les caractères `T[i]` et `M[j]` sont différents mais que les `j` précédents sont identiques. On compare alors `T[i]` avec le caractère de `M` qui suit le plus grand préfixe de `M` qui est un suffixe de `M[:j]`, c'est-à-dire l'élément de `M` d'indice `F[j-1]`**

II. 3 - Algorithme utilisant la structure de liste

Une autre possibilité pour chercher un motif dans une chaîne de caractères (ou séquence d'ADN) est de construire une liste contenant tous les sous-motifs de notre chaîne, triés par ordre alphabétique, puis de faire la recherche dans cette liste.
Par exemple, à la chaîne 'CATCG', on peut lui associer la liste :
`['C', 'A', 'T', 'G', 'CA', 'AT', 'TC', 'CG', 'CAT', 'ATC', 'TCG', 'CATC', 'ATCG', 'CATCG']`
que l'on peut ensuite trier pour obtenir la liste :
`['A', 'AT', 'ATC', 'ATCG', 'C', 'CA', 'CAT', 'CATC', 'CATCG', 'CG', 'G', 'T', 'TC', 'TCG']`.
La première étape de cette méthode est donc de trier une liste.

16. Écrire une fonction `triinsertion` de tri par insertion d'une liste de nombres.

- **3 pts**

```python
def triinsertion(L) :
    ''triinsertion(L : list) -> None''
    n = len(L)
    for i in range(1,n):
        aux = L[i]
        j = i
        while (j > 0 and L[j-1] > aux) :
            L[j] = L[j-1]
            j = j - 1
        L[j] = aux
```

17. Comment peut-on adapter la fonction `triinsertion` à une liste de chaîne de caractères ?

- **1 pt : L'ordre lexicographique permet d'ordonner totalement des chaînes de caractères. Cet ordre est déjà implémenté en langage Python. Il n'y a donc rien à changer au scrip de la question précédente.**

Après avoir obtenu une liste triée, on peut faire une recherche dichotomique dans cette nouvelle liste.

18. Écrire une fonction recherchedichotomique() de recherche dichotomique dans une liste de nombres triés.
Quel est l'intérêt de ce type d'algorithme (on parlera de complexité) ?

- **4 pts : code**

```python
def recherchedichotomique(a, L) :
    ''recherchedichotomique(a : float, L : list) -> m : int''
    N = len(L)
    deb = 0
    fin = N - 1
    while fin - deb >= 0 :
        m = (deb + fin) // 2
        if L[m] == a :
            return m
        elif L[m] > a :
            fin = m - 1
        else :
            deb = m + 1
    return 'L'élément recherché n'est pas dans la liste'
```

- **1 pt : L'intérêt des algorithmes dichotomiques est d'avoir une complexité en $\underset{n\to+\infty}{\Theta}\big(\ln(n)\big)$ où n est la taille de la variable d'intérêt.**

II. 4 - Fonction de hachage et évaluation de polynôme

II.4.a Fonction de hachage, algorithme de Karp-Rabin

19. On ne considère que des motifs de taille 3. Que renvoie la fonction de hachage avec les motifs 'CCC', 'ACG', 'GAG' ? On détaillera les calculs.

- **1 pt :** $\texttt{h}(\texttt{'CCC'}) = 8$
- **1 pt :** $\texttt{h}(\texttt{'ACG'}) = 6$
- **1 pt :** $\texttt{h}(\texttt{'GAG'}) = 8$

Dans cette fonction de hachage, nous avons besoin de transformer un entier en base b en un entier en base 10. On remarque que l'on peut éventuellement faire ce calcul en évaluant un polynôme.

II.4.b Évaluation de polynôme, Algorithme de Hörner

Dans cette **sous-partie**, nous allons nous intéresser à l'évaluation d'un polynôme et de son coût lorsque l'on compte les multiplications, les additions et les affectations comme des opérations unitaires.

Soit $P(X) = \sum_{k=0}^{n} a_k X^k$ un polynôme, il sera représenté par la liste $[a_n, \ldots, a_0]$.

20. Écrire une fonction `eval` ayant pour paramètre un polynôme P (donc une liste de nombres $[a_n, \ldots, a_0]$) et un nombre b. Cette fonction doit renvoyer la valeur de P en b, c'est-à-dire calculant :

$$P(b) = \sum_{k=0}^{n} a_k b^k$$

- **2 pts**

```
def eval(P, b) :
    ''eval(P : list, b : float) -> S : float''
    n = len(P) - 1
    S = 0
    for k in range(n+1) :
        S = S + P[n-k] * (b**k)
    return S
```

En admettant que le calcul de b^k utilise $k-1$ multiplications, on trouve une complexité quadratique. On peut être plus astucieux en utilisant l'algorithme de Hörner qui se base sur l'égalité suivante :

$$P(X) = \Big(\big(\cdots\big((a_n X + a_{n-1})\, X + a_{n-2}\big)\, X + \cdots\big)\, X + a_1\Big)\, X + a_0$$

Plus précisément, pour évaluer P en b, on commence par calculer $a_n \times b + a_{n-1}$, puis on multiplie le résultat par b et on ajoute a_{n-2}, etc. On trouvera alors une complexité linéaire.

21. Écrire une fonction itérative `hornerit` ayant pour paramètres un polynôme P, sous forme de liste, ainsi qu'un réel b, et renvoyant $P(b)$ en utilisant l'algorithme de Hörner.

- **2 pts**

```
def hornerit(P, b) :
    ''hornerit(P : list, b : float) -> y : float''
    n = len(P) - 1
    y = P[0]
    for k in range(1, n+1) :
        y = y * b + P[k]
    return y
```

22. Compléter la fonction `hornerrec` pour avoir une fonction récursive qui évalue un polynôme en utilisant l'algorithme de Hörner.

- **2 pts : 1 pt pour l'initialisation, 1 pt pour la ligne 8**

```
1  def hornerrec(P, b) :
2      ''hornerrec(P : list, b : float) -> float''
3      if len(P) == 1 :
4          return P[0]
5      else :
6          s = P[len(P) - 1]
7          s1 = P[0 : len(P) - 1]
8          return hornerrec(s1, b) * b + s
```