DS1

- **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

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

Démonstration.

```
1  def ind_max(L) :
2      '' ind_max(L : list) -> i_max : int''
3      n = len(L)
4      i_max = 0
5      for k in range(n) :
6          if L[i] > L[i_max] :
7              i_max = i
8      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`.

Démonstration.

```
1  import numpy as np
2
3  def zeroColSousDiag(j, M, N) :
4      '' zeroColSousDiag(j : int, M : np.matrix, N : np.matrix)''
5          ''-> [G : np.matrix, D : np.matrix]''
6      G = np.copy(M)
7      D = np.copy(N)
8      n = M.shape[0]
9      for i in range(j+1, n) :
10         D[i,:] = G[j,j] * D[i,:] - G[i,j] * D[j,:]
11         G[i,:] = G[j,j] * G[i,:] - G[i,j] * G[j,:]
12     return [G, D]
```

□

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

Démonstration.
Soit $n \in \mathbb{N}^*$. Soient `M` et `N` des matrices `numpy` carrées d'ordre n.
On considère que les opérations élémentaires sont la somme et le produit.

- À chaque tour de boucle de la structure itérative commençant en ligne 9, on effectue $3 \times 2 = 6$ opérations élémentaires (d'après les lignes 10 et 11).
- On effectue $(n-1) - (j+1) + 1 = n - j - 1$ tours de boucles. On effectue donc, dans le pire cas : $n - 0 - 1 = n - 1$ tours de boucles.

Ainsi, dans le pire cas, $6\,(n-1)$ opérations sont effectuées dans ce programme.

On en déduit que la complexité de l'algorithme est en $\underset{n \to +\infty}{\Theta}(n)$.

□

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.

Démonstration.

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

□

Problème

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]` ?

Démonstration.

- La commande `seq[3]` renvoie la chaîne de caractères contenant l'élément d'indice 3 de la chaîne de caractères `seq`.

caractère	A	T	C	G	T	A	C	G	T	A	C	G
indice	0	1	2	3	4	5	6	7	8	9	10	11

La commande `seq[3]` renvoie donc `'G'`.

- De plus, la commande `seq[2:6]` renvoie la chaîne de caractères contenant les éléments d'indice 2 à $6-1=5$ de la chaîne de caractères `seq`.

La commande `seq[2:6]` renvoie donc la chaîne de caractères `'CGTA'`.

Commentaire

On note `C` une chaîne de caractères. On rappelle les commandes suivantes.
- `C[i]` renvoie le coefficient d'indice `i` de la chaine de caractère `C` si `i` $\in [\![0, \texttt{len(C)} - 1]\!]$; renvoie `IndexError` sinon
- `C[i:j]` renvoie une nouvelle chaîne de caractères (copie superficielle) constituée de tous les éléments de `C` dont l'indice est un entier compris entre `i` et `j-1`.

□

Les fonctions que nous allons construire par la suite devront prendre en paramètre une chaîne de caractères ne contenant que des 'A', 'C', 'G' et 'T' (ceci correspond à une séquence d'ADN). Nous allons commencer par construire aléatoirement une séquence d'ADN.
Pour générer aléatoirement une séquence d'ADN composée de n caractères, on utilisera le principe suivant.

a) On commence par créer une chaîne de caracères vide.

b) Puis on tire aléatoirement n chiffres compris entre 1 et 4 et :
- si on obtient un 1, alors on ajoute un 'A' à notre chaîne de caractères ;
- si on obtient un 2, alors on ajoute un 'C' à notre chaîne de caractères ;
- si on obtient un 3, alors on ajoute un 'G' à notre chaîne de caractères ;
- si on obtient un 4, alors on ajoute un 'T' à notre chaîne de caractères.

c) On renvoie la chaîne de caractères ainsi construite.

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

Démonstration.

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

Commentaire

On pouvait également utiliser une liste plutôt qu'un dictionnaire. Cela aurait donné le script suivant.

```
1  def generation(n) :
2      seq = ''
3      codage = ['A', 'C', 'G', 'T']
4      for k in range(n) :
5          seq = seq + codage[rd.randint(0,4)]
6      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**.

Démonstration.

- **Début de la fonction**
  On commence par préciser la structure de la fonction :
  - × cette fonction se nomme `mystere`,
  - × elle prend en entrée un paramètre `seq` qui est une chaîne de caractères ,
  - × elle renvoie une liste.

```
1  def mystere(seq) :
```

```
17      return [a*100/len(seq), b*100/len(seq), c*100/len(seq), d*100/len(seq)]
```

En ligne 2, les variables `a`, `b`, `c` et `d` sont toutes initialisées à 0.

```
2  a, b, c, d = 0, 0, 0, 0
```

En ligne 3, la variable `i` est initialisée à l'indice du dernier caractère de la chaîne `seq`.

```
3      i = len(seq) - 1
```

- **Structure itérative**
  - × Les lignes 4 à 16 consistent à déterminer le nombre d'occurences des caractères 'A', 'C', 'G', 'T' sans la chaîne de caractères seq.
    Le nombre d'occurences
    - du caractère 'A' sera stocké dans la variable a,
    - du caractère 'C' sera stocké dans la variable b,
    - celui du caractère 'G' sera stocké dans la variable c,
    - celui du caractère 'T' sera stocké dans la variable d.

    Pour cela, l'énoncé choisit de parcourir la chaîne de caractère seq du dernier caractère au premier caractère. Cela est réalisé en mettant en place une structure itérative (boucle while).

```
4        while i >= 0 :
```

  - × On met ensuite en place une structure conditionnelle. Ici, quatre cas se présentent :
    - si le caractère d'indice i de seq est 'A', alors :
      - ▸ on incrémente la variable a de 1 pour signifier que le nombre d'occurence du caractère 'A' augmente de 1
      - ▸ on retranche 1 à la variable i pour examiner le caractère précédent dans la chaîne seq au prochain tour de boucle.

```
5            if seq[i] == 'A' :
6                a += 1
7                i -= 1
```

    - si le caractère d'indice i de seq est 'C', alors :
      - ▸ on incrémente la variable b de 1 pour signifier que le nombre d'occurence du caractère 'C' augmente de 1
      - ▸ on retranche 1 à la variable i pour examiner le caractère précédent dans la chaîne seq au prochain tour de boucle.

```
5            if seq[i] == 'C' :
6                b += 1
7                i -= 1
```

    - si le caractère d'indice i de seq est 'G', alors :
      - ▸ on incrémente la variable c de 1 pour signifier que le nombre d'occurence du caractère 'G' augmente de 1
      - ▸ on retranche 1 à la variable i pour examiner le caractère précédent dans la chaîne seq au prochain tour de boucle.

```
5            if seq[i] == 'G' :
6                c += 1
7                i -= 1
```

- sinon, le caractère d'indice `i` de `seq` est `'T'`, alors :
  - on incrémente la variable `d` de 1 pour signifier que le nombre d'occurence du caractère `'T'` augmente de 1
  - on retranche 1 à la variable `i` pour examiner le caractère précédent dans la chaîne `seq` au prochain tour de boucle.

```
5               else :
6                   d += 1
7                   i -= 1
```

- **Fin de la fonction**
  À l'issue de la boucle `while`, les variables `a`, `b`, `c` et `d` contiennent respectivement le nombre d'occurences des caractères `'A'`, `'C'`, `'G'` et `'T'` dans la chaîne `seq`.
  La liste présente en ligne 17 contient donc les fréquences, exprimées en pourcentage, des caractères `'A'`, `'C'`, `'G'` et `'T'` dans la chaîne `seq`.

```
17      return [a*100/len(seq), b*100/len(seq), c*100/len(seq), d*100/len(seq)]
```

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.

Commentaire

- Afin de permettre une bonne compréhension des mécanismes en jeu, on a détaillé la réponse à cette question. Cependant énoncer directement ce que permet de faire la fonction citée démontre la bonne compréhension du script et permet d'obtenir la totalité des points alloués à cette question.
- Il paraît curieux de ne pas avoir choisi de factoriser la command « `i -= 1` » en fin de boucle while. Cela aurait permis d'alléger et rendre plus lisible le code de cette fonction. On obtiendrait le script suivant.

```
1   def mystere(seq) :
2       a, b, c, d = 0, 0, 0, 0
3       i = len(seq) - 1
4       while i >= 0 :
5           if seq[i] == 'A' :
6               a += 1
7           elif seq[i] == 'C' :
8               b += 1
9           elif seq[i] == 'G' :
10              c += 1
11          else :
12              d += 1
13          i -= 1
14      return [a*100/len(seq), b*100/len(seq), c*100/len(seq), d*100/len(seq)]
```

Commentaire

- On peut s'interroger sur la construction de cette fonction. En effet, l'utilisation d'une simple boucle `for` aurait permis d'obtenir un script plus lisible. L'utilisation, dans ce contexte, de la structure itérative `while` tient de l'obfuscation de programme. En effet, ce choix rend artificiellement plus complexe le code de la fonction `mystere`. Voici ce que donnerait un script utilisant une structure itérative `for`.

```
def mystere(seq) :
    a, b, c, d = 0, 0, 0, 0
    n = len(seq)
    for i in range(n) :
        if seq[i] == 'A' :
            a += 1
        elif seq[i] == 'C' :
            b += 1
        elif seq[i] == 'G' :
            c += 1
        else :
            d += 1
    return [(a/n)*100, (b/n)*100, (c/n)*100, (d/n)*100]
```

- Enfin, il était possible, sur ce script, de proposer une écrire plus idiomatique de **Python** en utilisant un dictionnaire.

```
def mystere(seq) :
    n = len(seq)
    dico_occurence = { 'A' : 0, 'C' : 0, 'G' : 0, 'T' : 0 }
    for caract in seq :
        dico_occurence[caract] += 1
    freq = [(dico_occurence[c] / n) * 100 for c in dico_occurence.keys()]
    return freq
```

□

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).

Démonstration.

- Soit $n \in \mathbb{N}^*$. Soit `seq` une chaîne de n caractères.
  On considère que les opérations élémentaires sont la comparaison et l'addition.
  - × À chaque tour de boucle, on effectue :
    - dans le pire cas : 3 comparaisons. C'est le cas où le caractère d'indice `i` de la chaîne `seq` est `'T'`.
    - 2 additions : une pour mettre à jour l'une des variables `a`, `b`, `c` et `d`, et une pour mettre à jour la variable `i`.

    Ainsi, on effectue au plus 5 opérations élémentaires par tour de boucle.
  - × La variable `i` est initialisée à $n-1$ et la boucle `while` s'arrête lorsque `i` atteint -1. On effectue donc des tours de boucles pour `i` variant de $n-1$ à 0, c'est-à-dire : $(n-1)-0+1 = n$ tours de boucles.

  Finalement, on effectue $5n$ opérations élémentaires en tout.

La complexité de la fonction `mystere` est en $\underset{n\to+\infty}{\Theta}(n)$.

- La variable `i` 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.

  La variable permettant de démontrer la terminaison de l'algorithme est la variable `i`.

□

Partie II - Recherche d'un motif

On considère une chaîne de caractères `S = 'ACTGGTCACT'`, on appelle sous-chaîne de caractères de `S` une suite de caractères incluse dans `S`. Par exemple, `'TGG'` est une sous-chaîne de `S` mais `'TAG'` n'est pas une sous-chaîne de `S`.

Objectif
Rechercher une sous-chaîne de caractères `M` de longueur m appelée motif dans une chaîne de caractères `S` de longueur n

Il s'agit d'une problématique classique en informatique, qui répond aux besoins de nombreuses applications.
On trouve plus de 100 algorithmes différents pour cette même tâche, les plus célèbres datant des années 1970, mais plus de la moitié ont moins de 10 ans.

Dans cette **partie**, nous allons d'abord nous intéresser à l'algorithme naïf (**sous-partie II.1**), puis à deux autres algorithmes : l'algorithme de Knuth-Morris-Pratt (**sous-partie II.2**), et un algorithme utilisant une structure de liste (**sous-partie II.3**) et enfin, aux fonctions de hachage (**sous-partie II.4**).

Les différentes sous-parties sont indépendantes.

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.

Démonstration.

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

Commentaire

- On pouvait aussi proposer un script qui n'utilise pas le slicing, mais un peu moins naturelle.

```
def recherche(M, S) :
    n = len(S)
    m = len(M)
    for i in range(n-m+1) :
        j = 0
        while j < m and S[i+j] == M[j] :
            j += 1
        if j == m :
            return i
    return -1
```

- On pouvait également proposer un script utilisant une fonction auxiliaire pour tester si deux chaînes de caractères sont égales.

```
def egalite_str(S1, S2) :
    ''egalite_str(S1 : str, S2 : str) -> bool''
    n = len(S1)
    m = len(S2)
    if n != m :
        return False
    for k in range(n) :
        if S1[k] != S2[k] :
            return False
    return True
```

```
def recherche(M, S) :
    n = len(S)
    m = len(M)
    for k in range(n-m+1) :
        if egalite_str(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 $3 \cdot 10^9$ caractères. En combien de temps un ordinateur réalisant 10^{12} opérations par seconde fait-il ce calcul ?

Démonstration.

- Soit $(m, n) \in \left(\mathbb{N}^*\right)^2$.
  Soit M un motif de m caractère. Soit S une chaîne de n caractères.
  On considère que l'opération élémentaire est la comparaison.
  - × Dans le pire cas, le motif M ne se trouve pas dans la chaîne S mais les $m-1$ premiers caractères de M apparaissent bien dans S. Un exemple de pire cas est par exemple :

$$S = \underbrace{AAAAA\cdots AAA}_{n \text{ caractères}} \quad \text{et} \quad M = \underbrace{AAAAA\cdots AAT}_{m-1 \text{ caractères}}$$

× Dans un tel cas, pour chaque tour de boucle `for` (ligne 5), on effectue m comparaisons (il faut comparer chacun des caractères du motif M).
× On effectue de plus tous les tours de la boucle `for` : $n - m + 1$.

Finalement, on effectue $m \times (n - m + 1) = nm - m^2 + 1 = \Theta(nm)$ opérations élémentaires.

> Pour un motif de $m = 50$ caractères dans une séquence d'ADN de $n = 3 \cdot 10^9$ caractères, il faut effectuer de l'ordre de $nm = 15 \cdot 10^{10}$ opérations.

- Si un ordinateur réalise 10^{12} opérations par seconde, il effectue la recherche du motif de 50 caractères en $\dfrac{15 \times 10^{10}}{10^{12}}$ secondes. Or :

$$\frac{15 \times 10^{10}}{10^{12}} \;=\; 15 \times 10^{10-12} \;=\; 15 \times 10^{-2}$$

> Un ordinateur effectuera la recherche du motif en approximativement $0,15$ secondes.

Commentaire

Afin de permettre une bonne compréhension des mécanismes en jeu, on a détaillé la réponse à cette question. Cependant, fournir directement la réponse correcte démontre la bonne compréhension et permet d'obtenir la totalité des points alloués à cette question. □

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 ?

Démonstration.

- Pour comparer deux séquences d'ADN, on commence par découper la première séquence en morceaux de 50 caractères. Comme une séquence est composée de $3 \cdot 10^9$ caractères, on obtient donc : $\dfrac{3 \times 10^9}{50}$ motifs à comparer. Or :

$$\frac{3 \times 10^9}{50} \;=\; \frac{300 \times 10^7}{50} \;=\; \frac{30}{5} \times 10^7 \;=\; 6 \times 10^7$$

> Il faut donc chercher $6 \cdot 10^7$ motifs dans la seconde séquence d'ADN.

- D'après la question précédente, la comparaion d'un motif de 50 caractères s'effectue en $15 \cdot 10^{-2}$ secondes. Ainsi, la comparaison de tous les motifs de la première séquence d'ADN s'effectue en $(6 \times 10^7) \times (15 \times 10^{-2})$ secondes. Or :

$$(6 \times 10^7) \times (15 \times 10^{-2}) \;=\; 90 \times 10^5 \;=\; 9 \times 10^6$$

La comparaison de 2 séquences d'ADN nécessite de l'ordre de 9×10^6 secondes, soit $\dfrac{9 \times 10^6}{3600}$ heures. Or :

$$\frac{9 \times 10^6}{3600} \;=\; \frac{\cancel{9} \times 10^6}{\cancel{9} \times 4 \times 10^2} \;=\; \frac{1}{4} \times 10^4 \;=\; 2500$$

La comparaison de 2 séquences d'ADN s'effectue donc en environ 2500 heures, soit $\dfrac{2500}{24}$ jours. Autrement dit, environ 104 jours.

> La comparaison de 2 séquences d'ADN nécessite 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)

Lorsqu'un échec a lieu dans l'algorithme naïf, c'est-à-dire lorsqu'un caractère du motif est différent du caractère correspondant dans la séquence d'ADN, la recherche reprend à la position suivante en repartant au début du motif. Si le caractère qui a provoqué l'échec n'est pas au début du motif, cette recherche commence par comparer une partie du motif avec une partie de la séquence d'ADN qui a déjà été comparée avec le motif. L'idée de départ de l'algorithme de Knuth-Morris-Pratt est d'éviter ces comparaisons inutiles. Pour cela, une fonction annexe qui recherche le plus long préfixe d'un motif qui soit aussi un suffixe de ce motif est utilisée.

Avant d'étudier l'algorithme de Knuth-Morris-Pratt (**sous-partie II.2.b**) nous allons définir les notions de préfixe et suffixe (**sous-partie II.2.a**).

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'.

Démonstration.

Les préfixes du motif 'ACGTAC' sont : 'A', 'AC', 'ACG', 'ACGT' et 'ACGTA'.

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 ?

Démonstration.

- D'après la question précédente, le seul préfixe de 'ACGTAC' qui est aussi un de ses suffixes est 'AC'.

  Le plus grand préfixe de 'ACGTAC' qui soit aussi un suffixe est donc le motif 'AC'.

- On effectue le même détail qu'en question précédente pour la séquence 'ACAACA'.
  - × Les préfixes de 'ACAACA' sont : 'A', 'AC', 'ACA', 'ACAA' et 'ACAAC'.
  - × Les suffixes de 'ACAACA' sont : 'A', 'CA', 'ACA', 'AACA' et 'CAACA'.

  Le seul préfixe de 'ACAACA' qui est aussi un de ses suffixes est 'ACA'.

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

□

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

Nous rappelons que l'algorithme de Knuth-Morris-Pratt (KMP) est une fonction de recherche qui utilise une fonction annexe prenant en argument une chaîne de caractères M dont on notera la longueur m. Cette fonction annexe, appelée `fonctionannexe`, doit permettre, pour chaque lettre à la position `i`, de trouver le plus grand sous-mot de M qui finit par la lettre `M[i]` (c'est donc le plus grand suffixe de `M[:i+1]`) qui soit aussi un préfixe de M.
Le code de `fonctionannexe` se trouve à la question 13 du **DR 7**.

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

Démonstration.

- La variable de sortie de la fonction `fonctionannexe` est la variable `F`.

```
16        return F
```

- Cette variable est initialisée à la liste à 1 élément : l'entier 0.

```
2         F = [0]
```

- Elle est mise à jour dans la structure itérative `while` :
  - × soit en lui concaténant un entier `j + 1`,

```
7                 F.append(j + 1)
```

  - × soit en lui concaténant l'entier 0.

```
14                F.append(0)
```

- La variable `F` contient donc, à l'issue de la structure itérative, une liste d'entiers.

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.

Démonstration.

- Tout d'abord, la variable `m` n'est pas définie au sein de la fonction. Il faut donc effectuer ce stockage entre la ligne 1 et la ligne 2.

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

- Ensuite, on rappelle que :
  - × l'opérateur `=` sert à stocker un objet dans une variable,
  - × l'opérateur `==` sert à tester si deux variables sont égales.

  Ainsi, dans une structure conditionnelle, c'est toujours l'opérateur `==` qui est utilisé (et non l'opérateur `=`). Il faut donc modifier la ligne 6 de la façon suivante.

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

On obtient finalement le scrip suivant.

```
1  def fonctionannexe(M) :
2      ''fonctionannexe(M : str) -> F : list(int)''
3      m = len(M)
4      F = [0]
5      i = 1
6      j = 0
7      while i < m :
8          if M[i] == M[j] :
9              F.append(j + 1)
10             i = i + 1
11             j = j + 1
12         else :
13             if j > 0 :
14                 j = F[j - 1]
15             else :
16                 F.append(0)
17                 i = i + 1
18     return F
```

□

14. Décrire l'exécution de la fonction `fonctionannexe` lorsque $\texttt{M} = \texttt{'ACAACA'}$ 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`.

Démonstration.

- **Initialisation**
  Comme rappelé dans le Document Réponse :

  $$\texttt{i} = 1, \qquad \texttt{j} = 0 \qquad \text{et} \qquad \texttt{F} = [0]$$

  Notons de plus que, lorsque `M` est la chaîne `'ACAACA'`, alors la variable `m` contient la valeur 6 (nombre de caractères de la chaîne `M`).
- **Premier passage dans la boucle** `while`
  - × Avant le 1er passage dans la boucle, la variable `i` contient l'entier 1. Ainsi, la condition `i < m` est bien vérifiée ($1 < 6$). On entre donc dans le 1er tour de boucle.
  - × Comme, à ce stade, `i` contient la valeur 1 et `j` contient la valeur 0, on teste ensuite si la condition `M[1] == M[0]` est vraie. Or :

    $$\texttt{M[1]} = \texttt{'C'} \qquad \text{et} \qquad \texttt{M[0]} = \texttt{'A'}$$

    La condition de la ligne 6 n'est donc pas vérifiée. On passe donc aux instructions qui suivent la ligne 10.

× On teste alors si la condition `j > 0` est vérifiée. Ce n'est pas le cas car, à ce stade, la variable `j` contient l'entier 0. On passe donc aux instructions qui suivent la ligne 13.

- La variable `F` est mise à jour en lui concaténant, à droite, l'entier 0. Ainsi, après la ligne 14, la variable `F` contient la liste `[0, 0]`.
- La variable `i` est mise à jour en étant incrémentée de 1. Ainsi, après la ligne 15, la variable `i` contient l'entier $1 + 1 = 2$.
- La variable `j` n'est pas mise à jour.

Finalement, à l'issue du premier tour de boucle `while` :
`i = 2`, `j = 0` et `F = [0, 0]`.

- **Deuxième passage dans la boucle `while`**

× Avant le $2^{\text{ème}}$ passage dans la boucle, la variable `i` contient l'entier 2. Ainsi, la condition `i < m` est bien vérifiée ($2 < 6$). On entre donc dans le $2^{\text{ème}}$ tour de boucle.

× Comme, à ce stade, `i` contient la valeur 2 et `j` contient la valeur 0, on teste ensuite si la condition `M[2] == M[0]` est vraie. Or :

$$\texttt{M[2]} = \texttt{'A'} \qquad \text{et} \qquad \texttt{M[0]} = \texttt{'A'}$$

La condition de la ligne 6 est donc vérifiée. On passe donc aux instructions qui suivent cette ligne 6.

- La variable `F` est mise à jour en lui concaténant, à droite, l'entier `j + 1` $= 0 + 1 = 1$. Ainsi, après la ligne 7, la variable `F` contient la liste `[0, 0, 1]`.
- La variable `i` est mise à jour en étant incrémentée de 1. Ainsi, après la ligne 8, la variable `i` contient l'entier $2 + 1 = 3$.
- La variable `j` est mise à jour en étant incrémentée de 1. Ainsi, après la ligne 9, la variable `j` contient l'entier $0 + 1 = 1$.

Finalement, à l'issue du deuxième tour de boucle `while` :
`i = 3`, `j = 1` et `F = [0, 0, 1]`.

- **Troisième passage dans la boucle `while`**

× Avant le $3^{\text{ème}}$ passage dans la boucle, la variable `i` contient l'entier 3. Ainsi, la condition `i < m` est bien vérifiée ($3 < 6$). On entre donc dans le $3^{\text{ème}}$ tour de boucle.

× Comme, à ce stade, `i` contient la valeur 3 et `j` contient la valeur 1, on teste ensuite si la condition `M[3] == M[1]` est vraie. Or :

$$\texttt{M[3]} = \texttt{'A'} \qquad \text{et} \qquad \texttt{M[1]} = \texttt{'C'}$$

La condition de la ligne 6 n'est donc pas vérifiée. On passe donc aux instructions qui suivent la ligne 10.

× On teste alors si la condition `j > 0` est vérifiée. C'est bien le cas car, à ce stade, la variable `j` contient l'entier 1. On passe donc aux instructions qui suivent la ligne 11.

- La variable `F` n'est pas mise à jour.
- La variable `i` n'est pas mise à jour.
- La variable `j` est mise à jour en lui affectant l'entier `F[j - 1]`, c'est-à-dire `F[0]`, car `j` contient 1. Ainsi, comme à ce stade `F = [0, 0, 1]`, alors `F[0]` contient 0. On affecte donc à la variable `j` l'entier 0.

Finalement, à l'issue du troisième tour de boucle `while` :
`i = 3`, `j = 0` et `F = [0, 0, 1]`.

- **Quatrième passage dans la boucle** `while`
  - × Avant le $4^{\text{ème}}$ passage dans la boucle, la variable `i` contient l'entier 3. Ainsi, la condition `i < m` est bien vérifiée ($3 < 6$). On entre donc dans le $4^{\text{ème}}$ tour de boucle.
  - × Comme, à ce stade, `i` contient la valeur 3 et `j` contient la valeur 0, on teste ensuite si la condition `M[3] == M[0]` est vraie. Or :

    $$\texttt{M[3]} = \texttt{'A'} \qquad \text{et} \qquad \texttt{M[0]} = \texttt{'A'}$$

    La condition de la ligne 6 est donc vérifiée. On passe donc aux instructions qui suivent cette ligne 6.
    - La variable `F` est mise à jour en lui concaténant, à droite, l'entier $\texttt{j} + 1 = 0 + 1 = 1$. Ainsi, après la ligne 7, la variable `F` contient la liste `[0, 0, 1, 1]`.
    - La variable `i` est mise à jour en étant incrémentée de 1. Ainsi, après la ligne 8, la variable `i` contient l'entier $3 + 1 = 4$.
    - La variable `j` est mise à jour en étant incrémentée de 1. Ainsi, après la ligne 9, la variable `j` contient l'entier $0 + 1 = 1$.

    Finalement, à l'issue du quatrième tour de boucle `while` : `i = 4`, `j = 1` et `F = [0, 0, 1, 1]`.

- **Cinquième passage dans la boucle** `while`
  - × Avant le $5^{\text{ème}}$ passage dans la boucle, la variable `i` contient l'entier 4. Ainsi, la condition `i < m` est bien vérifiée ($4 < 6$). On entre donc dans le $5^{\text{ème}}$ tour de boucle.
  - × Comme, à ce stade, `i` contient la valeur 4 et `j` contient la valeur 1, on teste ensuite si la condition `M[4] == M[1]` est vraie. Or :

    $$\texttt{M[4]} = \texttt{'C'} \qquad \text{et} \qquad \texttt{M[1]} = \texttt{'C'}$$

    La condition de la ligne 6 est donc vérifiée. On passe donc aux instructions qui suivent cette ligne 6.
    - La variable `F` est mise à jour en lui concaténant, à droite, l'entier $\texttt{j} + 1 = 1 + 1 = 2$. Ainsi, après la ligne 7, la variable `F` contient la liste `[0, 0, 1, 1, 2]`.
    - La variable `i` est mise à jour en étant incrémentée de 1. Ainsi, après la ligne 8, la variable `i` contient l'entier $4 + 1 = 5$.
    - La variable `j` est mise à jour en étant incrémentée de 1. Ainsi, après la ligne 9, la variable `j` contient l'entier $1 + 1 = 2$.

    Finalement, à l'issue du cinquième tour de boucle `while` : `i = 5`, `j = 2` et `F = [0, 0, 1, 1, 2]`.

- **Sixième passage dans la boucle** `while`
  - × Avant le $6^{\text{ème}}$ passage dans la boucle, la variable `i` contient l'entier 5. Ainsi, la condition `i < m` est bien vérifiée ($5 < 6$). On entre donc dans le $6^{\text{ème}}$ tour de boucle.
  - × Comme, à ce stade, `i` contient la valeur 5 et `j` contient la valeur 2, on teste ensuite si la condition `M[5] == M[2]` est vraie. Or :

    $$\texttt{M[5]} = \texttt{'A'} \qquad \text{et} \qquad \texttt{M[2]} = \texttt{'A'}$$

    La condition de la ligne 6 est donc vérifiée. On passe donc aux instructions qui suivent cette ligne 6.
    - La variable `F` est mise à jour en lui concaténant, à droite, l'entier $\texttt{j} + 1 = 2 + 1 = 3$. Ainsi, après la ligne 7, la variable `F` contient la liste `[0, 0, 1, 1, 2, 3]`.

- La variable `i` est mise à jour en étant incrémentée de 1. Ainsi, après la ligne 8, la variable `i` contient l'entier $5+1=6$.
- La variable `j` est mise à jour en étant incrémentée de 1. Ainsi, après la ligne 9, la variable `j` contient l'entier $2+1=3$.

Finalement, à l'issue du cinquième tour de boucle `while` :
`i = 6`, `j = 3` et `F = [0, 0, 1, 1, 2, 3]`.

Commentaire

- Ce n'était pas demandé par l'énoncé, mais notons que, pour la chaîne `M = 'ACAACA'`, la boucle `while` s'arrête après ce $6^{\text{ème}}$ tour de boucle.
  En effet, avant le $7^{\text{ème}}$ passage dans la boucle, la variable `i` contient l'entier 6. Ainsi, la condition `i < m` n'est pas vérifiée ($6 \not< 6$). On n'entre donc pas dans le $7^{\text{ème}}$ tour de boucle.
- Une fois sorti de la structure itérative `while`, la fonction renvoie la valeur de la variable `F`. Ainsi, l'appel `fonctionannexe(M)` renvoie `[0, 0, 1, 1, 2, 3]`.

□

Explications de la fonction

L'énoncé est ici un peu avare en explications sur les mécanismes mis en jeu dans la fonction `fonctionannexe`. Essayons de mieux les comprendre.

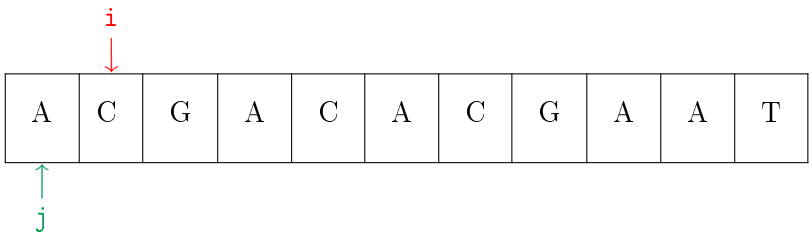
- On note toujours `m` la longueur du motif `M` et on travaillera plus précisément sur le motif `M` suivant :

$$\texttt{M} = \texttt{'ACGACACGAAT'}$$

(ici $\texttt{m} = 11$)

- Tout d'abord, comme le précise l'énoncé, pour tout $\texttt{i} \in [\![0, m-1]\!]$, l'élément `F[i]` doit contenir, à l'issue de la boucle `while`, le plus grand préfixe du motif `M` qui est aussi un suffixe de `M[: i+1]`. Plus précisément, `F[i]` contient le **nombre** de caractères de ce plus grand préfixe. Notons que cela suffit à caractériser complètement ce préfixe. On peut s'en convaincre sur l'exemple choisi dans cette remarque avec $\texttt{i} = 4$.
  - × On remarque : $\texttt{M[: i + 1]} = \texttt{M[: 5]} = \texttt{'ACGAC'}$.
  - × Mettons que l'on sache : $\texttt{F[i]} = \texttt{F[4]} = 2$. On sait donc que le plus grand préfixe de `M` qui est aussi un suffixe de `M[: i+1]` est de longueur 2. Comme c'est un préfixe de `M`, il s'agit forcément du sous-motif `M[:2]`, c'est-à-dire `'AC'`.
  - × Connaître la longueur du plus grand préfixe de `M` qui est aussi un suffixe de `M[: i+1]` permet donc de connaître exactement le préfixe en question. On se contente donc de stocker dans `F` la longueur de ce préfixe plus que le préfixe lui-même.
- Le programme utilise ensuite 2 curseurs :
  - × le curseur `i` qui va parcourir un à un tous les indices de 1 à $\texttt{m} - 1$. Plus précisément, à chaque mise à jour de la variable `F`, on incrémente `i` de 1 pour « passer » à l'indice suivant.
  - × le curseur `j` qui va parcourir le début du motif `M`. Nous détaillerons l'évolution de ce curseur plus loin.

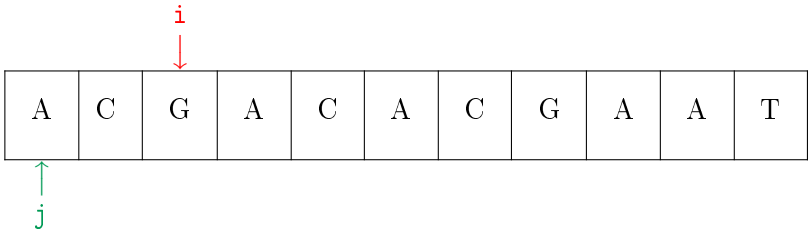
  Initialement, les curseurs sont positionnés ainsi.



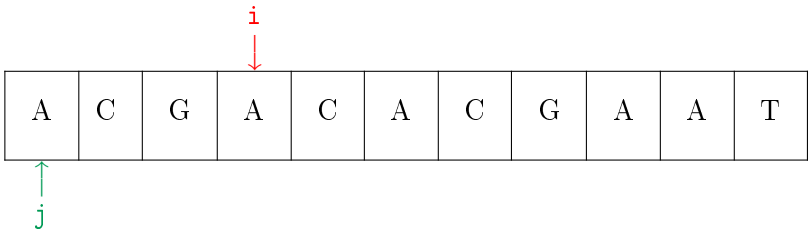
- Détaillons les évolutions des curseurs et de la liste `F`.
  - × 1er tour de boucle :
    Les caractères pointés par les curseurs `i` et `j` sont distincts.
    - Il n'y a donc pas de préfixe de `M` qui est un suffixe de `M[: i+1] = M[:1]`. On met donc à jour `F` en lui concaténant 0 (le plus grand préfixe de `M` qui est un suffixe de `M[:1]` contient 0 lettre).

      $$\texttt{F} = [0, 0]$$

      (on rappelle que la variable `F` était initialisée à `[0]`)
    - On décale le curseur `i` de 1 pour chercher le plus grand préfixe de `M` qui est un suffixe de `M[:2]`.



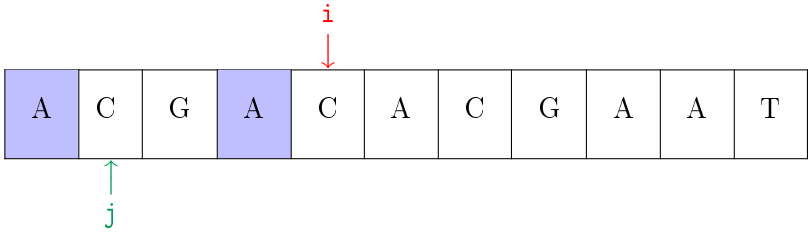
  - × 2ème tour de boucle
    Pour les mêmes raisons qu'au tour précédent, on obtient : `F = [0, 0, 0]` et :



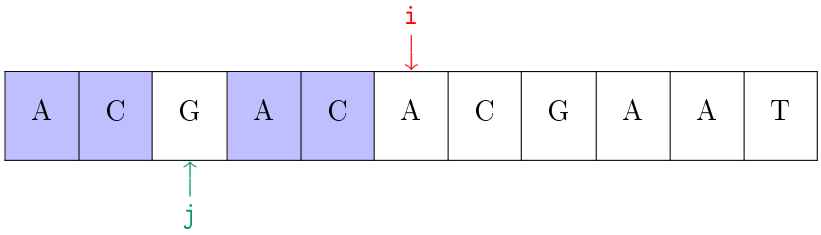
  - × 3ème tour de boucle
    Les caractères pointés par les curseurs `i` et `j` sont identiques (pour la 1ère fois).
    - Le plus grand préfixe de `M` qui est un suffixe de `M[: i+1] = M[:3]` est donc de longueur 1 (on pourrait démontrer en raisonnant par l'absurde qu'il ne peut pas être de longueur strictement supérieure à 1, sinon on aurait trouvé un préfixe de `M` qui est un suffixe de `M[:2]` de longueur au moins 1 à l'étape précédente, ce qui n'est pas le cas)

      $$\texttt{F} = [0, 0, 0, 1]$$

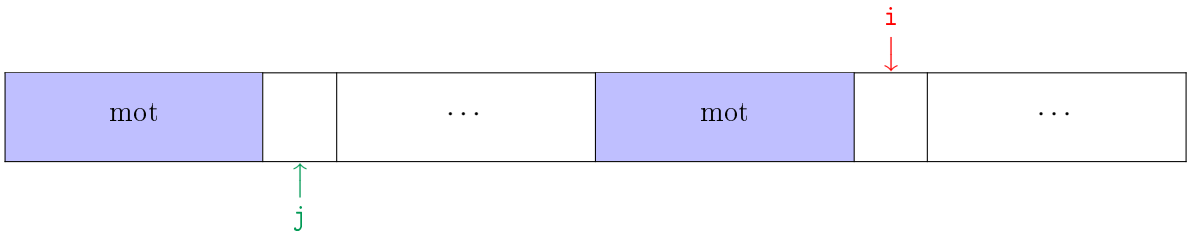
    - On incrémente alors les curseurs `i` et `j` de 1 pour voir si on peut trouver un préfixe plus grand à l'étape suivante.



× 4ème tour de boucle
Les caractères pointés par les curseurs `i` et `j` sont identiques. En raisonnant comme dans le cas précédent, on obtient : `F = [0, 0, 0, 1, 2]` et :

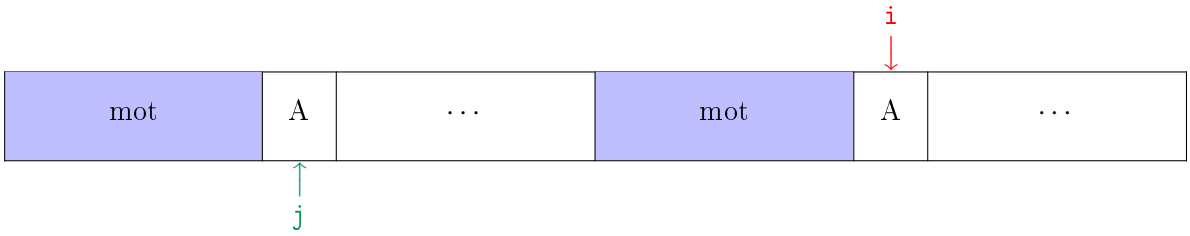


- Avant de poursuivre l'exemple, essayons maintenant de dégager le comportement général de la fonction. Au début du `i`ème tour de boucle, on est dans une configuration du type suivant.



Deux cas se présentent alors :

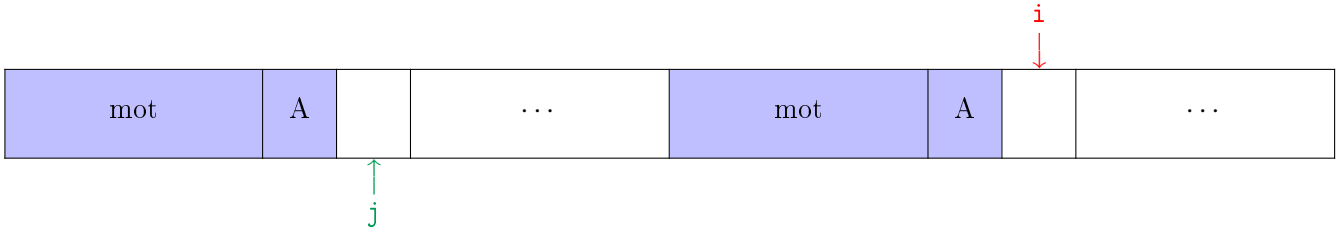
▶ soit les caractères `M[i]` et `M[j]` sont identiques.



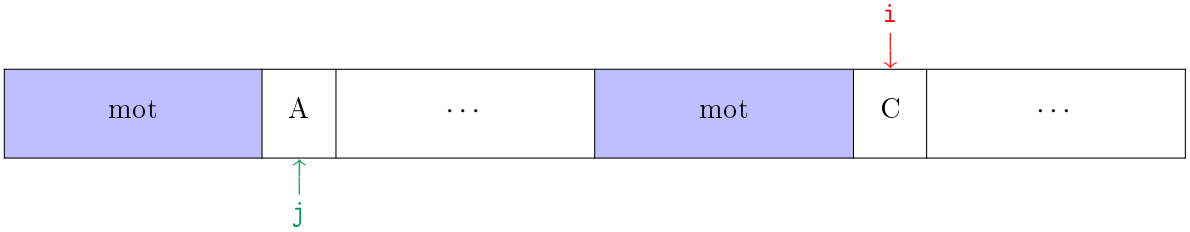
On note ℓ la longueur de `mot`. Dans ce cas :

- Le plus grand préfixe de `M` (donc de `M[: i+1]`) qui est un suffixe de `M[: i+1]` est donc de longueur $\ell + 1$. En effet :
  × le plus grand préfixe ne peut pas être de longueur strictement inférieure à $\ell + 1$ par construction : la chaîne de caractères `mot + 'A'`, de longueur $\ell + 1$, est bien un préfixe de `M` qui est aussi un suffixe de `M[: i+1]`.
  × le plus grand préfixe ne peut pas être de longueur strictement supérieure à $\ell+1$. Démontrons le en raisonnant par l'absurde.
  Supposons que le plus grand préfixe de `M[: i+1]` qui est aussi un suffixe de `M[: i+1]` est de longueur strictement plus grande que $\ell + 1$.
  Alors le plus grand préfixe qui est aussi un suffixe de `M[: i]` est de longueur strictement plus grande que ℓ. Ceci est absurde car on trouvé à l'étape précédente que ce plus plus grand préfixe est de longueur exactement ℓ.

  On concatène alors $\ell + 1$ à `F` : `F` $= [\cdots, \ell + 1]$.
- On incrémente alors les curseurs `i` et `j` de 1 pour voir si on peut trouver un préfixe plus grand à l'étape suivante.

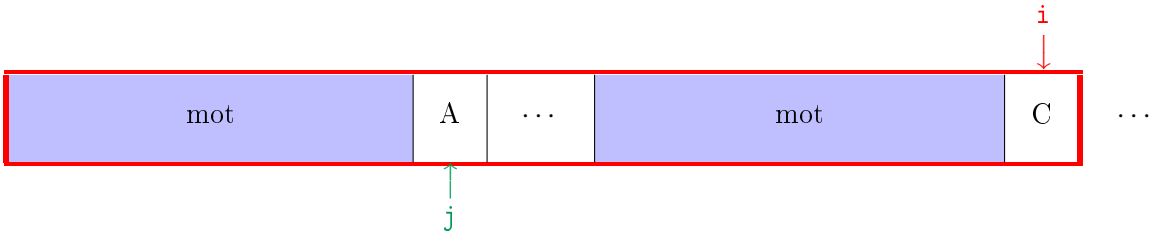


- soit les caractères `M[i]` et `M[j]` sont différents.

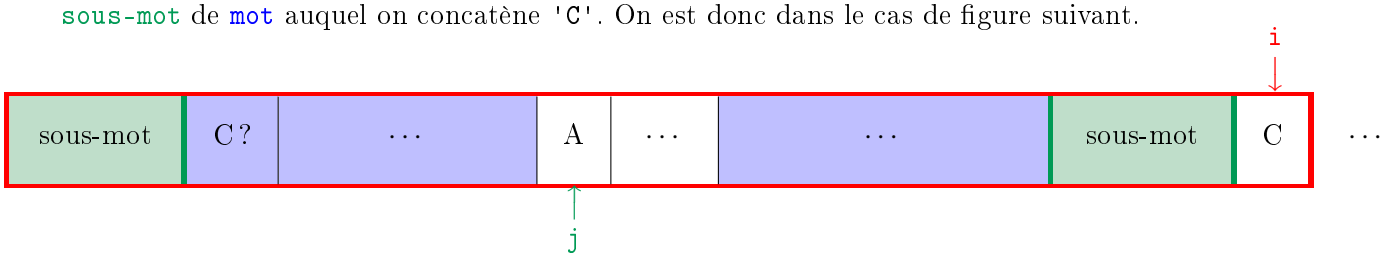


On note toujours ℓ la longueur de `mot`. Dans ce cas :

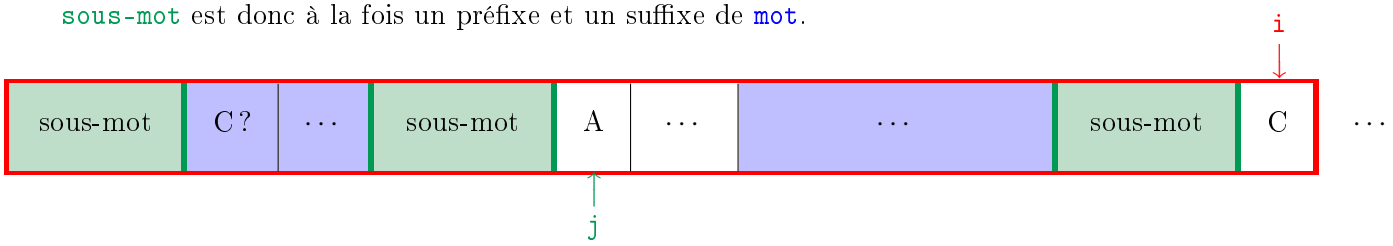
- la chaîne `mot + 'A'` n'est pas à la fois un préfixe et un suffixe de `M[: i+1]`. Il faut donc « ajuster » le curseur `j` de sorte à pouvoir en trouver un convenable.
  En « zoomant » sur les parties d'intérêt, on se trouve à chercher le plus grand préfixe qui est un suffixe de `M[: i+1]` (encadré en rouge) :



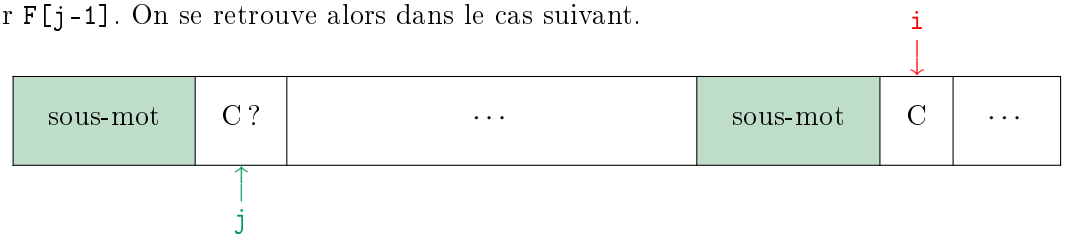
- Tout d'abord, notons que ce plus grand préfixe ne peut pas être de longueur strictement supérieure à $\ell + 1$. On peut toujours le démontrer en raisonnant par l'absurde.
  Supposons que le plus grand préfixe, qui est aussi un suffixe, de `M[: i+1]` est de longueur strictement plus grande que $\ell + 1$.
  Alors le plus grand préfixe, qui est aussi un suffixe, de `M[: i]` est de longueur strictement plus grande que ℓ. Ceci est absurde car on trouvé à l'étape précédente que ce plus grand préfixe est de longueur exactement ℓ.
- Comme ce préfixe n'est pas non plus de longueur $\ell + 1$ (car `M[j]` $\neq$ `M[i]`), alors on en cherche un de longueur inférieure ou égale à ℓ.
- Pour récapituler, on cherche la plus grande chaîne qui est un préfixe et un suffixe de `M[: i+1]`, de longueur inférieure ou égale à ℓ. Comme c'est un suffixe de `M[ : i+1]`, il s'écrit comme un `sous-mot` de `mot` auquel on concatène `'C'`. On est donc dans le cas de figure suivant.



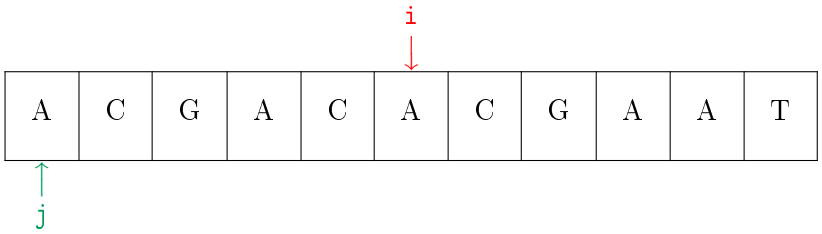
- Comme, par construction, la chaîne `mot` apparaît aux 2 endroits colorés en bleu, la chaîne `sous-mot` est donc à la fois un préfixe et un suffixe de `mot`.



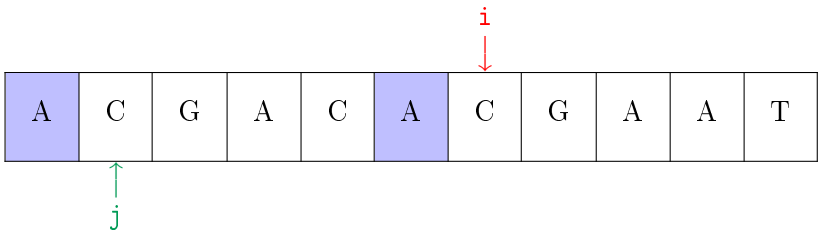
- Ainsi, on cherche un préfixe de `mot` qui est aussi un suffixe de `mot`.
  Or, comme on le voit sur les représentations graphiques précédentes, la chaîne `mot` est exactement la chaîne `M[: j]`.
  Et enfin, puisque l'on cherche le **plus grand** préfixe de `M[: i+1]` qui est aussi un suffixe de `M[: i+1]`, on va choisir, pour la mise à jour de `j`, l'indice qui suit la taille du **plus grand** préfixe de `M[:j]` qui est aussi un suffixe de `M[:j]`.
  Or, la taille du plus grand préfixe de `M[:j]` qui est aussi un suffixe de `M[:j]` est stockée dans l'élément `F[j-1]` (par définition de `F`. On met donc à jour la variable `j` en remplaçant sa valeur par `F[j-1]`. On se retrouve alors dans le cas suivant.



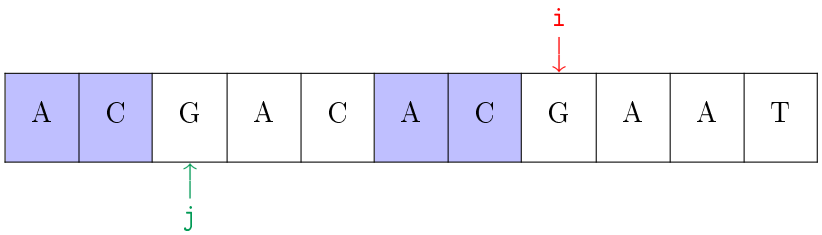
- On peut alors itérer le processus.
  En reprenant l'exemple de la chaîne `M = 'ACGACACGAAT'`, on obtient :
  × au 5ème tour de boucle
  Les caractères pointés par `i` et `j` sont distincts. On met donc seulement à jour la variable `j` en lui affectant `F[j-1] = F[2] = 0`



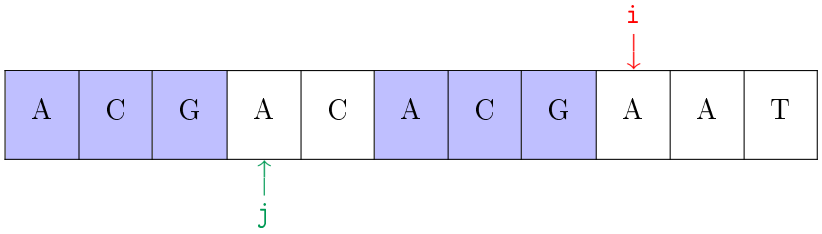
  × au 6ème tour de boucle
  Les caractères pointés par `i` et `j` sont identiques. On incrémente donc les variables `i` et `j` de 1, et on concatène à `F` la longueur du plus grand préfixe de `M` qui est un suffixe de `M[: i+1]`.
  On obtient : `F = [0, 0, 0, 1, 2, 1]`.



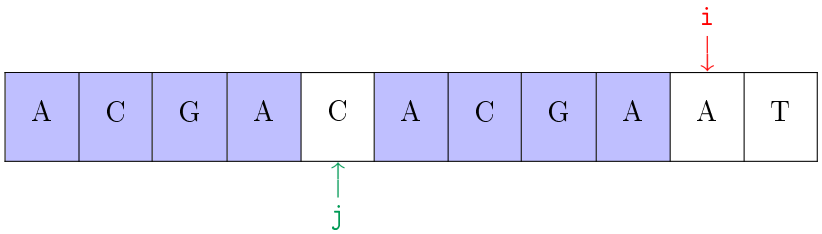
  × au 7ème tour de boucle
  Les caractères pointés par `i` et `j` sont identiques. On incrémente donc les variables `i` et `j` de 1, et on concatène à `F` la longueur du plus grand préfixe de `M` qui est un suffixe de `M[: i+1]`.
  On obtient : `F = [0, 0, 0, 1, 2, 1, 2]`.



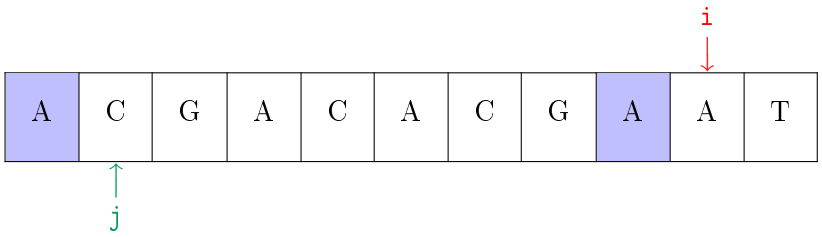
× au 8ème tour de boucle
Les caractères pointés par `i` et `j` sont identiques. On incrémente donc les variables `i` et `j` de 1, et on concatène à `F` la longueur du plus grand préfixe de `M` qui est un suffixe de `M[: i+1]`. On obtient : `F = [0, 0, 0, 1, 2, 1, 2, 3]`.



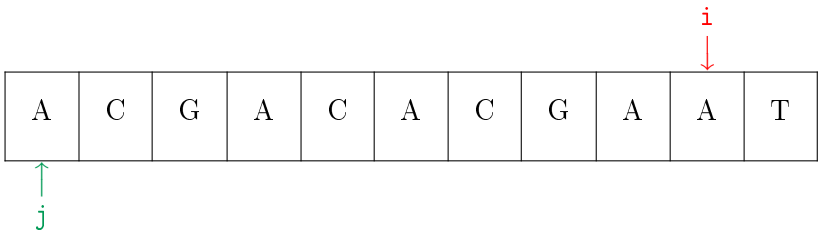
× au 9ème tour de boucle
Les caractères pointés par `i` et `j` sont identiques. On incrémente donc les variables `i` et `j` de 1, et on concatène à `F` la longueur du plus grand préfixe de `M` qui est un suffixe de `M[: i+1]`. On obtient : `F = [0, 0, 0, 1, 2, 1, 2, 3, 4]`.



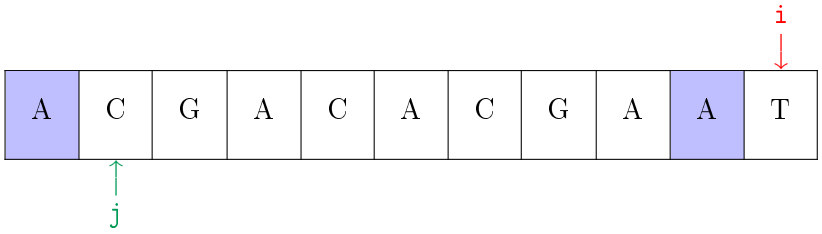
× au 10ème tour de boucle
Les caractères pointés par `i` et `j` sont distincts. On met donc seulement à jour la variable `j` en lui affectant `F[j-1] = F[3] = 1`.



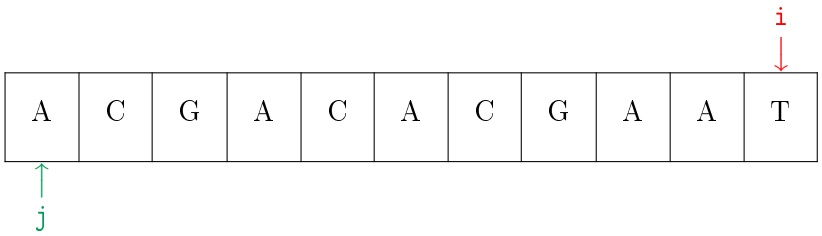
× au 11ème tour de boucle
Les caractères pointés par `i` et `j` sont distincts. On met donc seulement à jour la variable `j` en lui affectant `F[j-1] = F[0] = 0`.



× au 12ème tour de boucle
Les caractères pointés par `i` et `j` sont identiques.On incrémente donc les variables `i` et `j` de 1, et on concatène à `F` la longueur du plus grand préfixe de `M` qui est un suffixe de `M[: i+1]`. On obtient : `F = [0, 0, 0, 1, 2, 1, 2, 3, 4, 1]`.



× au 13ème tour de boucle
Les caractères pointés par `i` et `j` sont distincts. On met donc seulement à jour la variable `j` en lui affectant `F[j-1] = F[0] = 0`.



× au 14ème tour de boucle
Les caractères pointés par `i` et `j` sont distincts. De plus `j = 0`. On incrémente donc la variable `i` de 1 et on concatène 0 à `F`, ce qui fournit :

$$\texttt{F} = [0, 0, 0, 1, 2, 1, 2, 3, 4, 1, 0]$$

À l'issue de ce 14ème tour de boucle, la variable `i` contient la valeur 11. Ainsi : `i` $\not<$ `m`. La boucle `while` s'arrête donc, et la fonction renvoie la liste `F`.

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

Démonstration.

- D'après l'énoncé, la ligne 2 permet de stocker dans la variable `F` la liste où l'élément d'indice `i`ème contient le nombre de caractères du plus grand préfixe de `M` qui est un suffixe de `M[: i+1]`.
- La ligne 3 initialise la variable `i` à 0 et la ligne 4 initialise la variable `j` à 0.
- Les lignes correspondant au cas où on a trouvé le mot sont les lignes 7 et 8.
- Dans le cas où on a trouvé le mot, la fonction renvoie l'indice `i - j`, c'est-à-dire l'indice de première apparition du mot `M` dans la chaîne de caractère `T`.
- Les lignes correspondant au cas où on a trouvé deux lettres identiques sont les lignes 6 à 11. Deux cas se présentent alors :
  × soit on a trouvé le mot. On est alors ramené au cas précédent.
  × soit on n'a pas trouvé le mot. On poursuit alors la recherche en passant au caractère suivant à la fois dans la chaîne de caractère `M` et la chaîne de caractère `T`.

- Les lignes correspondant au cas où on a trouvé deux lettres différentes sont les lignes 12 à 16. Deux cas se présentent alors :
  - × soit `j` $= 0$. Ceci implique que l'on comparait `T[i]` au 1^{er} caractère de `M`. Comme ces caractères sont différents, on poursuit la recherche en passant au caractère suivant dans `T`.
  - × soit `j` > 0. Ceci implique 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]` (la justification est similaire à celle de la fonction `fonctionannexe` détaillée en question précédente).

□

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.

Démonstration.
On propose le script suivant.

```
1  def triinsertion(L) :
2      ''triinsertion(L : list) -> None''
3      n = len(L)
4      for i in range(1,n):
5          aux = L[i]
6          j = i
7          while (j > 0 and L[j-1] > aux) :
8              L[j] = L[j-1]
9              j = j - 1
10         L[j] = aux
```

Commentaire

- On rappelle que l'algorithme de tri par insertion ne renvoie aucune variable car la liste `L` est modifiée **en place** (c'est-à-dire au fil de l'algorithme).
- Il s'agit ici d'un algorithme du cours de $1^{\text{ère}}$ année qu'il faut maîtriser. Il en est de même des algorithmes de tri par sélection et de tri fusion.

□

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

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

```
1  def triinsertion(S) :
2      ''triinsertion(S : str) -> None''
3      n = len(S)
4      for i in range(1,n):
5          aux = S[i]
6          j = i
7          while (j > 0 and S[j-1] > aux) :
8              S[j] = S[j-1]
9              j = j - 1
10         S[j] = aux
```

□

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) ?

Démonstration.
On propose le script suivant.

```
1  def recherchedichotomique(a, L) :
2      ''recherchedichotomique(a : float, L : list) -> m : int''
3      N = len(L)
4      deb = 0
5      fin = N - 1
6      while fin - deb >= 0 :
7          m = (deb + fin) // 2
8          if L[m] == a :
9              return m
10         elif L[m] > a :
11             fin = m - 1
12         else :
13             deb = m + 1
14     return 'L'élément recherché n'est pas dans la liste'
```

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.

Commentaire

On peut déterminer explicitement la complexité de la fonction `recherchedichotomique`.
Soit $n \in \mathbb{N}$. Soit `L` une liste à n éléments.
On considère que l'opération élémentaire est la comparaison.

- On sait que :
  - × l'intervalle de recherche initial (l'ensemble d'indices $[\![0, n-1]\!]$) est de longueur n.
  - × la longueur de l'intervalle de recherche est divisée par 2 à chaque tour de boucle. À la fin du $i^{\text{ème}}$ tour de boucle, l'intervalle de recherche est donc de largeur $\left\lfloor \frac{n}{2^i} \right\rfloor$.
  - × on effectue au maximum 3 comparaisons par tours de boucle.
  - × l'algorithme s'arrête lorsque l'intervalle devient de longueur strictement inférieur à 1.

  On obtient le nombre de tours de boucle effectués en procédant par équivalence :

$$\begin{aligned} \frac{n}{2^i} < 1 &\Leftrightarrow \frac{2^i}{n} > 1 && \textit{(par stricte décroissance de la fonction inverse sur } \mathbb{R}_+^*\textit{)} \\ &\Leftrightarrow 2^i > n \\ &\Leftrightarrow i\ \ln(2) > \ln(n) && \textit{(par stricte croissance de la fonction } \ln \textit{ sur } \mathbb{R}_+^*\textit{)} \end{aligned}$$

  Ainsi, on effectue au plus $\left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil$ tours de boucle.

  Ainsi, on effectue dans le pire cas $3 \times \left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil$ opérations élémentaires.
- La complexité de l'algorithme de recherche dichotomique dans un tableau trié est donc en $\underset{n\to+\infty}{\Theta}\big(\ln(n)\big)$.

□

II. 4 - Fonction de hachage et évaluation de polynôme

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

Certains algorithmes, comme l'algorithme de Karp-Rabin (1987), utilisent une fonction de hachage h qui à un motif renvoie une valeur numérique.
Voici un exemple de fonction de hachage :

- à chaque caractère de l'alphabet, on associe une valeur. Ici, on va associer à 'A' la valeur 0, à 'C' la valeur 1, à 'G' la valeur 2 et à 'T' la valeur 3. Pour un motif de taille n, on obtient donc une suite de chiffre $a_{n-1} \dots a_1 a_0$. Par exemple, à la chaîne 'TAGC', on lui associe la suite de chiffre 3021 ;
- cette suite de chiffre est considérée comme l'écriture d'un entier en base b, où b est le nombre de caractères présents dans l'alphabet. On a donc ici $b = 4$;
- on calcule ensuite cet entier en base 10 (on calcule donc $a_{n-1}b^{n-1} + \dots + a_1 b^1 + a_0 b^0$) ;
- puis on calcule le reste de la division euclidienne de ce nombre par 13.

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.

Démonstration.

- Cas du motif `'CCC'`.
  - × On commence par associer à la chaîne `'CCC'` la suite de chiffres 111.
  - × On considère que cette suite de chiffres est l'écriture d'un entier en base 4.
  - × On calcule cet entier en base 10.

$$1 \times 4^2 + 1 \times 4^1 + 1 \times 4^0 \;=\; 16 + 4 + 1 \;=\; 21$$

  - × On calcule le reste dans la division euclidienne de 21 par 13. On remarque :

$$\begin{cases} 21 \;=\; 13 \times 1 + 8 \\ 0 \leqslant 8 < 13 \end{cases}$$

  Le reste dans la division euclidienne de 21 par 13 est donc 8.

  Pour le motif `'CCC'`, la fonction de hachage renvoie 8.

- Cas du motif `'ACG'`.
  - × On commence par associer à la chaîne `'ACG'` la suite de chiffres 012.
  - × On considère que cette suite de chiffres est l'écriture d'un entier en base 4.
  - × On calcule cet entier en base 10.

$$0 \times 4^2 + 1 \times 4^1 + 2 \times 4^0 \;=\; 0 + 4 + 2 \;=\; 6$$

  - × On calcule le reste dans la division euclidienne de 6 par 13. On remarque :

$$\begin{cases} 6 \;=\; 13 \times 0 + 6 \\ 0 \leqslant 6 < 13 \end{cases}$$

  Le reste dans la division euclidienne de 6 par 13 est donc 6.

  Pour le motif `'ACG'`, la fonction de hachage renvoie 6.

- Cas du motif `'GAG'`.
  - × On commence par associer à la chaîne `'GAG'` la suite de chiffres 202.
  - × On considère que cette suite de chiffres est l'écriture d'un entier en base 4.
  - × On calcule cet entier en base 10.

$$2 \times 4^2 + 0 \times 4^1 + 2 \times 4^0 \;=\; 32 + 0 + 2 \;=\; 34$$

  - × On calcule le reste dans la division euclidienne de 34 par 13. On remarque :

$$\begin{cases} 34 \;=\; 13 \times 2 + 8 \\ 0 \leqslant 8 < 13 \end{cases}$$

  Le reste dans la division euclidienne de 34 par 13 est donc 8.

  Pour le motif `'GAG'`, la fonction de hachage renvoie 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 $[a_n, \dots, a_0]$.

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

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

Démonstration.
On propose la fonction suivante.

```
1  def eval(P, b) :
2      ''eval(P : list, b : float) -> S : float''
3      n = len(P) - 1
4      S = 0
5      for k in range(n+1) :
6          S = S + P[n-k] * (b**k)
7      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.

Démonstration.
On propose la fonction suivante.

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

□

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

Démonstration.
On complète la fonction du Document Réponse de la façon suivante.

```
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
```

□