Les candidats sont informés que la précision des raisonnements algorithmiques ainsi que le soin apporté à la rédaction et à la présentation des copies seront des éléments pris en compte dans la notation. Il convient en particulier de rappeler avec précision les références des questions abordées. Si, au cours de l'épreuve, un candidat repère ce qui peut lui sembler être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Remarques générales :

- ✓ *L'épreuve se compose de deux parties indépendantes.*
- ✓ *Toutes les instructions et les fonctions demandées seront écrites en Python.*
- ✓ *Les questions non traitées peuvent être admises pour aborder les questions ultérieures.*
- ✓ *Toute fonction peut être décomposée, si nécessaire, en plusieurs fonctions.*

~ ~

Partie I : Calcul numérique

Résolution numérique d'équation différentielle

Les méthodes d'Adams-Bashforth et d'Adams-Moulton

Dans cette partie, on suppose que les modules suivants sont importés :

from numpy import *

from matplotlib.pyplot import *

Les équations différentielles (**ED**) apparaissent très souvent dans la modélisation de la physique et des sciences de l'ingénieur. Trouver la solution d'une **ED** ou d'un système d'**ED** est ainsi un problème courant, souvent difficile ou impossible à résoudre de façon analytique. Il est alors nécessaire de recourir à des méthodes numériques pour les résoudre.

Le **problème de Cauchy** consiste à trouver une fonction $y(t)$ définie sur l'intervalle $[a, b]$ telle que :

$$\begin{cases} y' = f(t, y(t)) \quad ; \quad \forall\, t \in [\,a\,,b\,] \\ y(a) = y_0 \end{cases}$$

Pour obtenir une approximation numérique de la solution **y(t)** sur l'intervalle **[a, b]**, nous allons estimer la valeur de cette fonction en un nombre fini de points t_i, pour **i = 0, 1, … , n**, constituants les nœuds du maillage. La solution numérique obtenue aux points t_i est notée $y_i = y(t_i)$. L'écart entre deux abscisses, noté **h**, est appelé : **le pas de discrétisation**.

Les principales méthodes de résolution numérique des **ED** sont séparées en deux grandes catégories :

- ✓ **les méthodes à un pas** : Le calcul de la valeur $\mathbf{y_{n+1}}$ au nœud $\mathbf{t_{n+1}}$ fait intervenir la valeur $\mathbf{y_n}$ obtenue à l'abscisse précédente. Les principales méthodes sont celles de : *Euler, Runge-Kutta, Crank-Nicholson …*
- ✓ **les méthodes à multiples pas** : Le calcul de la valeur $\mathbf{y_{n+1}}$ au nœud $\mathbf{t_{n+1}}$ fait intervenir plusieurs valeurs $\mathbf{y_n}$, $\mathbf{y_{n-1}}$, $\mathbf{y_{n-2}}$, **…** obtenues aux abscisses précédentes. Les principales méthodes sont celles de : *Nyström, Adams-Bashforth, Adams-Moulton, Gear …*

Les méthodes d'Adams-Bashforth et d'Adams-Moulton :

Ce sont des méthodes basées sur des techniques d'intégration numérique, qui utilisent les polynômes interpolateurs de Lagrange. La formulation générale de ces méthodes est :

$$y_{n+1} = y_n + h \sum_{j=-1 \text{ ou } 0}^{p} b_j \, f(t_{n-j}, y_{n-j})$$

Dans la suite de cette partie, nous nous intéressons aux méthodes d'*Adams-Bashforth* et d'*Adams-Moulton* d'ordre **2**.

- ❖ **Le schéma d'Adams-Bashforth à deux pas, explicite, d'ordre 2 :**

y_0 donné ;

y_1 calculé par la méthode Euler ;

$$y_{n+1} = y_n + \frac{h}{2}(3 f(t_n, y_n) - f(t_{n-1}, y_{n-1}))$$

- ❖ **Le schéma d'Adams-Moulton à un pas, implicite, d'ordre 2 :**

y_0 donné ;

$$y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y_{n+1}))$$

En pratique, les schémas d'**Adams-Moulton** ne sont pas utilisés comme tels, car ils sont implicites. On utilise plutôt conjointement un schéma d'**Adams-Moulton** et un schéma d'**Adams-Bashforth** de même ordre, pour construire un schéma *prédicteur-correcteur* : Dans le schéma d'**Adams-Moulton**, on utilise la valeur de $\boldsymbol{y_{n+1}}$ prédite (calculée) par le schéma d'**Adams-Bashforth**.

Le schéma prédicteur-correcteur d'ordre 2, d'Adams-Bashforth et Adams-Moulton est :

y_0 donné ;

y_1 calculé par la méthode d'Euler ;

$$y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, k))$$

avec $k = y_n + \frac{h}{2}(3 f(t_n, y_n) - f(t_{n-1}, y_{n-1}))$

Q. 1 : Écrire une fonction **ABM2 (*f*, a, b, y0, h)**, qui reçoit en paramètres :

- ✓ ***f*** est la fonction qui représente une équation différentielle du premier ordre ;
- ✓ **a** et **b** sont les deux bornes de l'intervalle d'intégration ;
- ✓ **y0** est la valeur initiale à l'instant **a** (y(a) = y0) ;
- ✓ **h** est le pas de discrétisation.

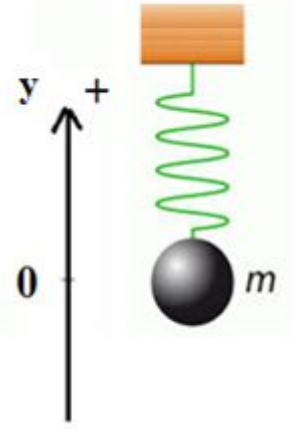
En utilisant le schéma prédicteur-correcteur d'ordre **2**, d'Adams-Bashforth et Adams-Moulton, la fonction renvoie **T** et **Y**, qui peuvent être des listes ou des tableaux :

- ➢ **T** contient la subdivision de l'intervalle **[a, b]**, en valeurs régulièrement espacée, utilisant le **pas** de discrétisation **h** ;
- ➢ **Y** contient les approximations des valeurs $\mathbf{y(t_i)}$, à chaque instant $\mathbf{t_i}$ de **T**.

Application à un oscillateur harmonique

Un objet, de masse ***m***, est suspendu à un ressort fixé à un support.

0 représente le point d'équilibre, et ***y(t)*** représente la position de l'objet par rapport au point d'équilibre, avec la direction positive vers le haut.



Si l'objet suspendu est tiré verticalement vers le bas, celui-ci effectue un mouvement harmonique. La forme générale de l'équation différentielle modélisant ce mouvement harmonique est :

$$\textbf{(E)} \qquad m \cdot \ddot{y}(t) + b \cdot \dot{y}(t) + k \cdot y(t) = F(t)$$

Les paramètres de cette équation différentielle sont :

- ✓ ***b*** : constante de proportionnalité de la force d'amortissement ;
- ✓ ***F(t)*** : force extérieure appliquée sur l'objet ;
- ✓ ***k*** : constante de rappel du ressort ;
- ✓ ***m*** : masse de l'objet.

Si $b = 0$ et $F(t) = 0$, alors le mouvement harmonique est dit : **simple**.

Si $b \neq 0$ et $F(t) = 0$, alors mouvement harmonique est dit: **amorti**.

Si $F(t) \neq 0$, alors mouvement harmonique est dit : **forcé**.

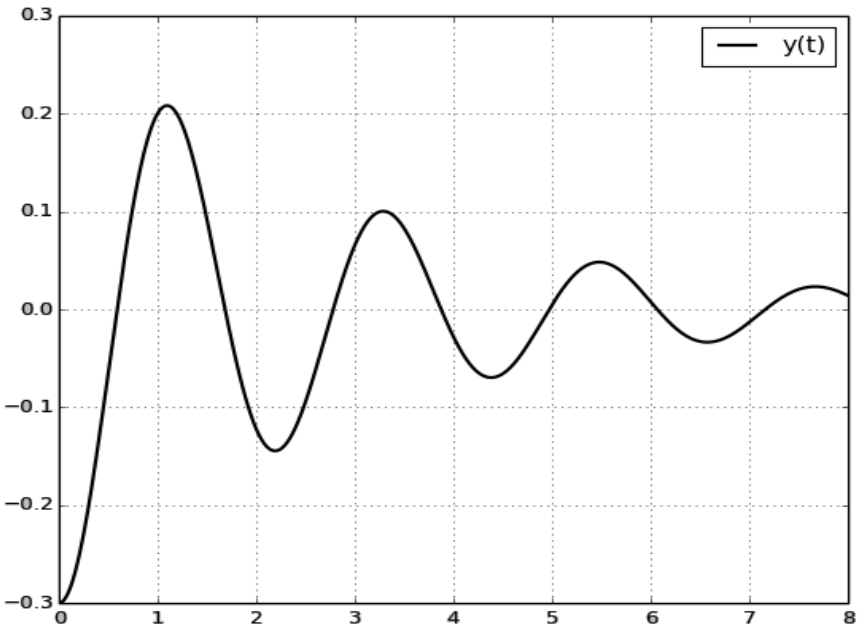
Q. 2 : On suppose que les valeurs des paramètres de l'équation différentielle **(E)**, sont :

$$b = 1 \quad ; \quad k = 12.5 \quad ; \quad m = 1.5 \quad et \quad F(t) = 0$$

Écrire l'équation différentielle **(E)** sous la forme **z' = G(t, z)**, avec **G(t, z)** une fonction à deux variables **(t, z)** dont **z** est un tableau de longueur **2.**

Q. 3 : On suppose avoir tiré l'objet suspendu verticalement vers le bas, avant de le relâcher *sans vitesse initiale*.

En utilisant le schéma prédicteur-correcteur d'ordre **2**, d'Adams-Bashforth et Adams-Moulton, écrire le code Python qui permet de tracer la courbe ci-dessous, représentant la position **y(t)** de l'objet **m** suspendu, toutes les $\mathbf{10^{-3}}$ **secondes**.



Q. 4 : Écrire la fonction **racines (P)**, qui reçoit en paramètre la liste (ou le tableau) **P** des positions **y(t)** de l'objet suspendu, à intervalles de temps de $\mathbf{10^{-3}}$ **secondes**. La fonction affiche les valeurs des zéros de la fonction **y(t)** : les valeurs des **t** tels que **y(t) = 0**, ou **y(t) ≈ 0** avec la précision $\mathbf{10^{-3}}$**.**

Exemple :

La fonction affiche les valeurs suivantes :

0.589 , 1.684 , 2.78 , 3.875 , 4.971 , 6.067 , 7.162

* *

Partie II :

Bioinformatique : Recherche d'un motif

La bioinformatique, c'est quoi?

Au cours des trente dernières années, la récolte de données en biologie a connu un boom quantitatif grâce notamment au développement de nouveaux moyens techniques servant à comprendre l'ADN et d'autres composants d'organismes vivants. Pour analyser ces données, plus nombreuses et plus complexes aussi, les scientifiques se sont tournés vers les nouvelles technologies de l'information. L'immense capacité de stockage et d'analyse des données qu'offre l'informatique leur a permis de gagner en puissance pour leurs recherches. La bioinformatique sert donc à stocker, traiter et analyser de grandes quantités de données de biologie. Le but est de mieux comprendre et mieux connaître les phénomènes et processus biologiques.

Dans le domaine de la bioinformatique, la recherche de sous-chaîne de caractères, appelée **motif**, dans un texte de taille très grande, était un composant important de beaucoup d'algorithmes. Puisqu'on travaille sur des textes de tailles très importantes, on a toujours cette obsession de l'efficacité des algorithmes : moins on a à faire de comparaisons, plus rapide sera l'exécution des algorithmes.

Le motif à rechercher, ainsi que le texte dans lequel on effectue la recherche, sont écrits dans un alphabet composé de quatre caractères : **'A'** , **'C'** , **'G'** et **'T'**. Ces caractères représentent les quatre bases de l'ADN : **A**dénine, **C**ytosine, **G**uanine et **T**hymine.

Codage des caractères de l'alphabet

L'alphabet est composé de quatre caractères seulement : **'A'**, **'C'**, **'G'** et **'T'.** Dans le but d'économiser la mémoire, on peut utiliser un codage binaire *réduit* pour ces quatre caractères.

Q. 1 : Quel est le *nombre minimum de bits* nécessaires pour coder quatre éléments ?

Q. 2 : Donner un exemple de code binaire pour chaque caractère de cet alphabet.

Préfixe d'une chaîne de caractères

*Un **préfixe** d'une chaîne de caractères **S** non vide, est une sous-chaîne non vide de **S**, composée de la suite des caractères entre la première position de **S** et une certaine position dans **S**.*

Exemples : S = 'TATCTAGCTA'

- ✓ La chaîne **'TAT'** est un préfixe de **'TAT**CTAGCTA' ;
- ✓ La chaîne **'TATCTA'** est un préfixe de **'TATCTA**GCTA' ;
- ✓ La chaîne **'TATCTAGCTA'** est un préfixe de **'TATCTAGCTA'** ;
- ✓ La chaîne **'T'** est un préfixe de **'T**ATCTAGCTA' ;
- ✓ La chaîne 'CTAGC' n'est pas un préfixe de **S**.

Q. 3 : Écrire une fonction **prefixe (M, S)**, qui reçoit en paramètres deux chaînes de caractères **M** et **S** non vides, et qui retourne **True** si la chaîne **M** est un préfixe de **S**, sinon, elle retourne **False**.

Q. 4 : Quel est le nombre de *comparaisons élémentaires* effectuées par la fonction **prefixe (M, S)**, dans le pire cas ? Déduire la complexité de la fonction **prefixe (M, S)**, dans le pire cas.

Suffixe d'une chaîne de caractères

*Un **suffixe** d'une chaîne de caractères **S** non vide, est une sous-chaîne non vide de **S**, composée de la suite des caractères, en partant d'une certaine position **p** dans **S**, jusqu'à la dernière position de **S**. L'entier **p** est appelé : **position du suffixe**.*

Exemples : S = 'TATCTAGCTA'

- ✓ La chaîne **'TCTAGCTA'** est un suffixe de 'TA**TCTAGCTA'**, sa position est : **2** ;
- ✓ La chaîne **'GCTA'** est un suffixe de 'TATCTA**GCTA'**, sa position est : **6** ;
- ✓ La chaîne **'TATCTAGCTA'** est un suffixe de **'TATCTAGCTA'**, sa position est : **0** ;
- ✓ La chaîne **'A'** est un suffixe de 'TATCTAGCT**A'**, sa position est : **9** ;
- ✓ La chaîne 'CTAGC' n'est pas un suffixe de **S**.

Q. 5 : Écrire une fonction **liste_suffixes (S)**, qui reçoit en paramètre une chaîne de caractères **S** non vide, et qui renvoie une liste de tuples. Chaque tuple de cette liste est composé de deux éléments : un suffixe de **S**, et la position **p** de ce suffixe dans **S** ($0 \leq p <$ taille de S).

Exemple : S = 'TATCTAGCTA'

La fonction **liste_suffixes (S)** renvoie la liste :

[('TATCTAGCTA', **0**), ('ATCTAGCTA', **1**), ('TCTAGCTA', **2**), ('CTAGCTA', **3**), ('TAGCTA', **4**), ('AGCTA', **5**), ('GCTA', **6**), ('CTA', **7**), ('TA', **8**), ('A', **9**)]

Tri de la liste des suffixes

Q. 6 : Écrire une fonction **tri_liste (L)**, qui reçoit en paramètre la liste **L** des suffixes d'une chaîne de caractères non vide, créée selon le principe décrit dans la question **5**. La fonction **tri_liste**, trie les éléments de **L** dans l'ordre alphabétique des suffixes.

NB : On ne doit pas utiliser la fonction sorted(), ou la méthode sort().

Exemple :

L = [('TATCTAGCTA', **0**), ('ATCTAGCTA', **1**), ('TCTAGCTA', **2**), ('CTAGCTA', **3**), ('TAGCTA', **4**), ('AGCTA', **5**), ('GCTA', **6**), ('CTA', **7**), ('TA', **8**), ('A', **9**)]

Après l'appel de la fonction **tri_liste (L)**, on obtient la liste suivante :

[('**A**', 9) , ('**AGCTA**', 5) , ('**ATCTAGCTA**', 1) , ('**CTA**', 7) , ('**CTAGCTA**', 3) , ('**GCTA**', 6) , ('**TA**', 8) , ('**TAGCTA**', 4) , ('**TATCTAGCTA**', 0) , ('**TCTAGCTA**', 2)]

Recherche dichotomique d'un motif :

La liste des suffixes **L** contient les tuples composés des suffixes d'une chaîne de caractères **S** non vide, et de leurs positions dans **S**. Si la liste **L** est triée dans l'ordre alphabétique des suffixes, alors, la recherche d'un motif **M** dans **S**, peut être réalisée par une simple *recherche dichotomique* dans la liste **L**.

Q. 7 : Écrire une fonction **recherche_dichotomique (M, L)**, qui utilise le *principe de la recherche dichotomique*, pour rechercher le motif **M** dans la liste des suffixes **L**, triée dans l'ordre alphabétique des suffixes : Si **M** est un préfixe d'un suffixe dans un tuple de **L**, alors la fonction retourne la position de ce suffixe. Si la fonction ne trouve pas le motif **M** recherché, alors la fonction retourne **None**.

Exemple :

La liste des suffixes de 'TATCTAGCTA', triée dans l'ordre alphabétique des suffixes est :

L = [('A', 9) , ('AGCTA', 5) , ('ATCTAGCTA', 1) , ('CTA', 7) , ('CTAGCTA', 3) , ('GCTA', 6) , ('TA', 8) , ('TAGCTA', 4) , ('TATCTAGCTA', 0) , ('TCTAGCTA', 2)]

- ✓ **recherche_dichotomique ('CTA', L)** renvoie la position **3** ; *'**CTA**' est préfixe de '**CTA**GCTA'*
- ✓ **recherche_dichotomique ('TA', L)** renvoie la position **0** ; *'**TA**' est préfixe de '**TA**TCTAGCTA'*
- ✓ **recherche_dichotomique ('CAGT', L)** renvoie **None**. *'CAGT' n'est préfixe d'aucun suffixe*

Q. 8 : On pose **m** et **k** les tailles respectives de la chaîne **M** et la liste **L**. Déterminer, en fonction de **m** et **k**, la complexité de la fonction **recherche_dichotomique (M, L).** Justifier votre réponse.

L'arbre des suffixes d'une chaîne de caractères :

L'arbre des suffixes, d'une chaîne de caractères **S** non vide, est un **arbre n-aire** qui permet de représenter tous les suffixes de **S**. Cet arbre possède les propriétés suivantes :

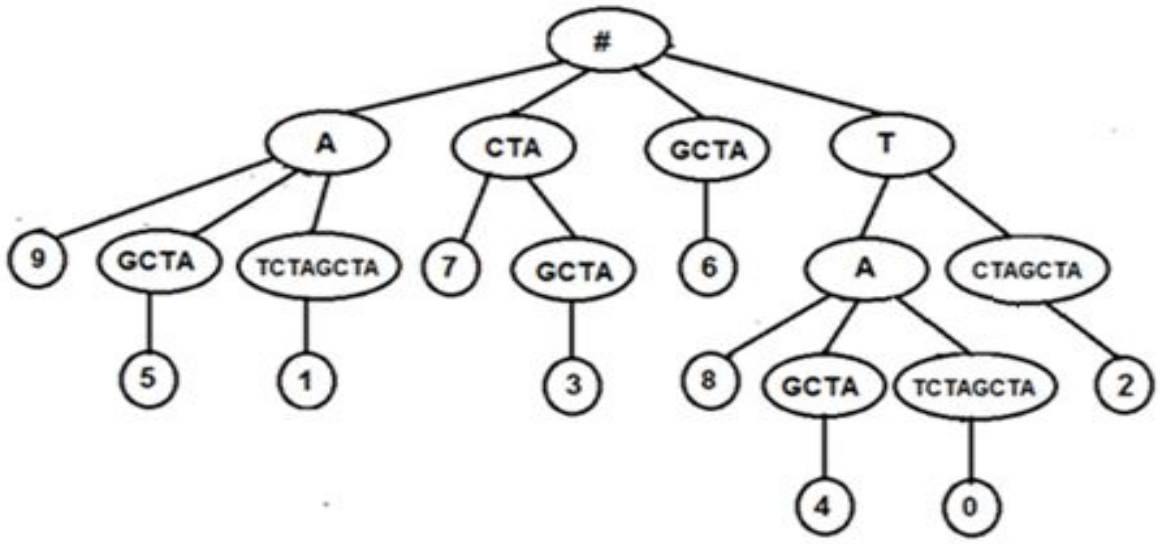
- ✓ La racine de l'arbre est marquée par le caractère '**#**' ;
- ✓ Chaque nœud de l'arbre contient une sous-chaîne de **S**, non vide ;
- ✓ Les premiers caractères, des sous-chaînes fils d'un même nœud, sont différents ;
- ✓ Chaque feuille de l'arbre contient un entier positif qui correspond à la position d'un suffixe dans la chaîne **S**.

Exemple : S = 'TATCTAGCTA'

La liste des suffixes de **S**, triée dans l'ordre alphabétique des suffixes est :

L = [('A', 9) , ('AGCTA', 5) , ('ATCTAGCTA', 1) , ('CTA', 7) , ('CTAGCTA', 3) , ('GCTA', 6) , ('TA', 8) , ('TAGCTA', 4) , ('TATCTAGCTA', 0) , ('TCTAGCTA', 2)]

L'arbre des suffixes est crée à partir de la liste des suffixes triée :



La racine de l'arbre '**#**' se trouve au niveau **0** de l'arbre, ses fils se trouvent au niveau **1**, … . En parcourant une branche de l'arbre depuis le niveau **1**, et en effectuant la concaténation des sous-chaînes des nœuds, on obtient un suffixe de la chaîne **S** ; et dans la feuille, on trouve la position de ce suffixe dans **S**.

Exemples :

- ✓ En parcourant les nœuds de la première branche à droite de l'arbre '**T**' + '**CTAGCTA**', on obtient le suffixe '**TCTAGCTA**'. La feuille contient la position **2** de ce suffixe ;
- ✓ En parcourant les nœuds de la troisième branche à partir de la droite de l'arbre '**T**' + '**A**' + '**GCTA**', on obtient le suffixe '**TAGCTA**'. La feuille contient la position **4** de ce suffixe.

Pour représenter l'arbre des suffixes, on utilise une liste composée de deux éléments : Le nœud, et une liste contenant tous les fils de ce nœud :

- ✓ Un nœud **N** est représenté par la liste : **[N , [f_1 , f_2 , … , f_n]**] ;
- ✓ Chaque fils f_k est représenté par la liste : **[f_k , [tous les fils de f_k]**] ;
- ✓ Une feuille **F** est représentée par la liste : **[F , [ ]**].

On suppose que la fonction **arbre_suffixes (S)**, reçoit en paramètre une chaîne de caractères **S** non vide, et construit et renvoie l'arbre des suffixes de **S**, représenté par une liste, selon le principe décrit ci-dessus.

Exemple :

arbre_suffixes ('TATCTAGCTA') renvoie la liste **R** suivante :

R = ['#', [['A', [[9, [ ]], ['GCTA', [[5,[ ]]]], ['TCTAGCTA', [[1, [ ]]]]]], ['CTA',[[7, [ ]], ['GCTA', [[3, [ ]]]]]], ['GCTA', [[6, [ ]]]], ['T', [['A', [[8, [ ]], ['GCTA', [[4, [ ]]]], ['TCTAGCTA', [[0, [ ]]]]]], ['CTAGCTA', [[2, [ ]]]]]]]

Q. 9 : Écrire une fonction **feuilles (R)**, qui reçoit en paramètre une liste **R** représentant l'arbre des suffixes d'une chaîne de caractères non vide, et qui retourne la liste contenant toutes les feuilles de l'arbre **R**.

Exemple : feuilles (R) revoie la liste **[9 , 5 , 1 , 7 , 3 , 6 , 8 , 4 , 0 , 2]**

L'intérêt, de l'arbre des suffixes, c'est qu'il permet d'effectuer une recherche de **toutes les occurrences** d'un motif dans **un temps qui dépend uniquement de la longueur du motif**, et non de la chaîne dans laquelle on effectue la recherche. En effet, la recherche d'un motif se réalise par la lecture du motif recherché et par le parcours simultané de l'arbre des suffixes. Trois cas peuvent survenir :

- ✓ **On se retrouve sur un nœud, et le motif est un préfixe de ce nœud :** Dans ce cas, les feuilles du sous arbre de ce nœud, correspondent aux positions de toutes les occurrences du motif dans la chaîne.
- ✓ **On se retrouve sur un nœud, et ce nœud est un préfixe du motif :** Dans ce cas, on continue la recherche parmi les fils de ce nœud.
- ✓ **On ne peut plus descendre dans l'arbre :** Dans ce cas, il n'y a pas d'occurrence du motif recherché dans la chaîne.

Q. 10 : Écrire une fonction **recherche_arbre (M, R)**, qui reçoit en paramètres un motif **M** non vide, et une liste **R** représentant l'arbre des suffixes d'une chaîne de caractères non vide. La fonction retourne une liste contenant les positions de toutes les occurrences de **M** dans cette chaîne de caractères.

Exemples :

R est la liste qui représente l'arbre des suffixes de la chaîne **S** = 'TATCTAGCTA'.

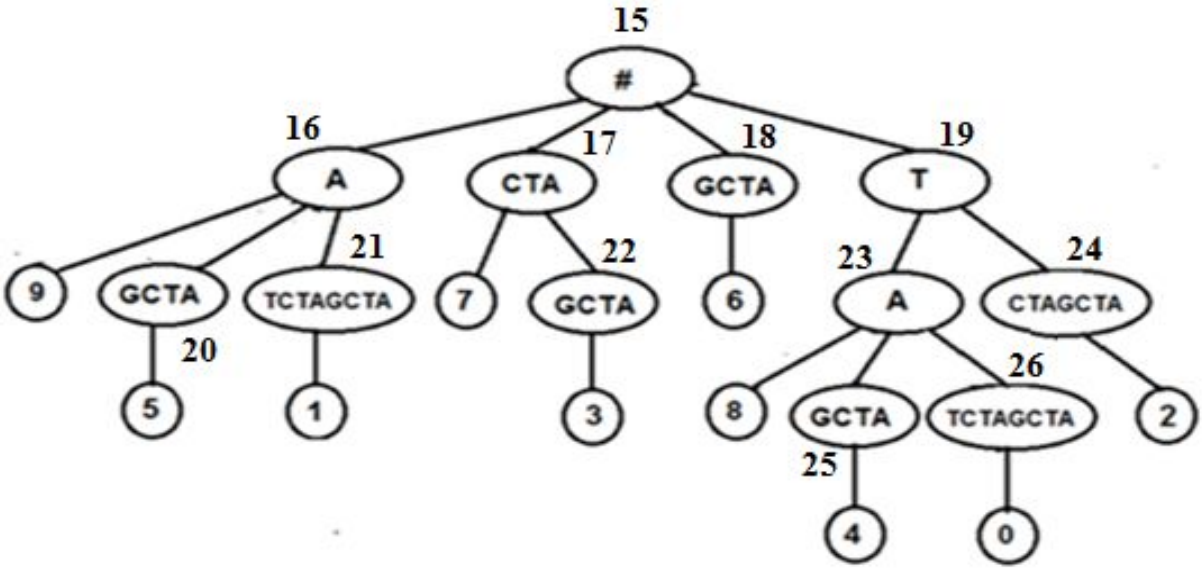
- ✓ **recherche_arbre** ('TA', R) renvoie la liste : **[8 , 4 , 0]**
- ✓ **recherche_arbre** ('TCTAG', R) renvoie la liste : **[2]**
- ✓ **recherche_arbre** ('CTA', R) revoie la liste : **[7 , 3]**
- ✓ **recherche_arbre** ('TACG', R) renvoie la liste : **[ ]**

Base de données d'un arbre des suffixes

Dans cette partie, on utilise une base de données pour stocker un arbre des suffixes d'une chaîne de caractères **S** non vide. Pour identifier les nœuds de l'arbre, on donne un code unique pour chaque nœud :

- Chaque nœud interne est identifié par un entier positif unique supérieur à la taille de **S** ;
- Les feuilles de l'arbre sont les différentes positions **p** dans la chaîne **S** ($0 \leq$ **p** < taille de **S**). Chaque feuille de l'arbre est identifiée par l'entier **p** qu'elle contient.

Exemple : Codes des nœuds internes de l'arbre des suffixes de la chaîne **S = 'TATCTAGCTA'**



Dans cet exemple, la taille de **S** est **10**. Chaque feuille est identifiée par un entier compris entre **0** et **9**, et qui correspond à un indice dans **S**. Chaque nœud interne est identifié par un entier positif unique supérieur à **10**. Le choix et la répartition des codes des nœuds sont effectués de façon aléatoire.

La base de données de l'arbre des suffixes est composée de deux tables : **Nœuds** et **Liens.**

a- Structure de la table **Nœuds** :

Nœuds (**code** (entier) , **nom** (texte), **niveau** (entier))

La table **Nœuds** contient les nœuds internes de l'arbre des suffixes, et elle est composée de trois champs :

- ✓ Le champ **code** contient un entier positif unique pour chaque nœud interne ;
- ✓ Le champ **nom** contient le nom du nœud interne ;
- ✓ Le champ **niveau** contient un entier positif qui représente le niveau du nœud interne dans l'arbre.

b- Structure de la table **Liens** :

Liens (**pere** (entier) , **fils** (entier))

La table **liens** contient toutes les liaisons *père-fils* dans l'arbre, elle est composée de deux champs :

- ✓ Le champ **pere** contient les codes des nœuds pères ;
- ✓ Le champ **fils** contient les codes des nœuds fils ;

c- Exemples d'enregistrements dans les deux tables :

Table : Nœuds		
code	***nom***	***niveau***
15	**'#'**	**0**
16	**'A'**	**1**
17	**'CTA'**	**1**
18	**'GCTA'**	**1**
19	**'T'**	**1**
20	**'GCTA'**	**2**
21	**'TCTAGCTA'**	**2**
…	**…**	**…**
26	**'TCTAGCTA'**	**3**

Table : Liens	
pere	***fils***
15	**16**
15	**17**
15	**18**
15	**19**
16	**9**
16	**20**
16	**21**
20	**5**
21	**1**
…	**…**

Q. 11 : Déterminer la *clé primaire* de la table **Nœuds**, et la *clé primaire* de la table **Liens**.

Justifier votre réponse.

Q. 12 : Écrire une requête SQL, qui renvoie **les noms des nœuds, sans répétition, tels que le motif 'GCTA' est un préfixe ou suffixe des noms de ces nœuds, triés dans l'ordre croissant du champ niveau.**

Q. 13 : Écrire une requête SQL, qui renvoie **la hauteur de l'arbre.**

Exemple : La hauteur de l'arbre exemple est **5**.

Q. 14 : Le **degré d'un nœud** est égal au nombre de fils de ce nœud.

Exemple : Le degré de la racine de l'arbre exemple est **4**.

Écrire une requête SQL, qui renvoie **les noms et les niveaux des nœuds dont le degré est supérieur à 2, triés dans l'ordre croissant selon le degré.**

Q. 15 : Le **degré d'un arbre** est égal au plus grand des degrés de ses nœuds.

Écrire une requête SQL, qui renvoie **le degré de l'arbre.**

Q. 16 : Écrire une requête SQL, qui renvoie **les feuilles de l'arbre ayant le niveau maximal.**

Q. 17 : Écrire une requête, en algèbre relationnelle, qui renvoie **les feuilles ayant le niveau supérieur à 2.**

Plus long motif commun

Le problème du **plus long motif commun** à deux chaînes de caractères **P** et **Q** non vides, consiste à déterminer le (ou les) plus long motif qui est en même temps sous-chaîne de **P** et de **Q**. Ce problème se généralise à la recherche du plus long motif commun à plusieurs chaînes de caractères.

Pour rechercher le plus long motif commun à deux chaînes de caractères **P** et **Q** non vides, on commence par créer une matrice **M** remplie par des zéros : Le nombre de lignes de **M** est égal à la taille de **P**, et le nombre de colonnes de **M** est égal à la taille de **Q**. Ensuite, on compare les caractères de **P** avec ceux de **Q**, et on écrit le nombre de caractères successifs identiques, dans la case correspondante de la matrice **M**.

Exemple : P = 'GCTAGCATT' et **Q** = 'CATTGTAGCT'

La matrice **M**, de comparaison des caractères de **P** avec ceux de **Q**, est la suivante :

	C	A	T	T	G	T	A	G	C	T
G	0	0	0	0	**1**	0	0	**1**	0	0
C	**1**	0	0	0	0	0	0	0	**2**	0
T	0	0	**1**	**1**	0	**1**	0	0	0	**3**
A	0	**1**	0	0	0	0	**2**	0	0	0
G	0	0	0	0	**1**	0	0	**3**	0	0
C	**1**	0	0	0	0	0	0	0	**4**	0
A	0	**2**	0	0	0	0	**1**	0	0	0
T	0	0	**3**	**1**	0	**1**	0	0	0	**1**
T	0	0	**1**	**4**	0	**1**	0	0	0	**1**

À partir de cette matrice **M**, On peut extraire les plus longs motifs communs à **P** et **Q** : '**CATT**' et '**TAGC**'

Q-18 : Écrire une fonction : **matrice (P, Q)**, qui reçoit en paramètres deux chaînes de caractères **P** et **Q** non vides, et qui retourne la matrice **M** de comparaison des caractères de **P** avec ceux de **Q**, selon le principe décrit dans l'exemple ci-dessus.

Exemple : P = 'GCTAGCATT' et Q = 'CATTGTAGCT'

matrice (P, Q) renvoie la matrice **M** suivante :

[[0, 0, 0, 0, 1, 0, 0, 1, 0, 0], [1, 0, 0, 0, 0, 0, 0, 0, 2, 0], [0, 0, 1, 1, 0, 1, 0, 0, 0, 3], […] , …]

Q-19 : Écrire une fonction **plus_long_mc (P, M)**, qui reçoit en paramètres une chaîne de caractères **P** non vide, et la matrice **M** de comparaison des caractères de la chaîne **P** avec ceux d'une autre chaîne **Q** non vide aussi, selon le principe décrit dans l'exemple ci-dessus. La fonction renvoie la liste, *sans doublons*, de tous les plus longs motifs communs à **P** et **Q**.

Exemple :

P = 'GCTAGCATT' et **M** est la matrice de comparaison des caractères de **P** avec ceux de la chaîne 'CATTGTAGCT'.

plus_long_mc (P, M) retourne la liste des plus longs motifs communs : **['TAGC' , 'CATT']**

Plus long motif commun dans un fichier

On considère le fichier texte dont le chemin absolu est : **'C:\génomes.txt'**. On suppose que ce fichier contient *au moins deux lignes*.

Q-20 : Écrire le code python qui permet d'afficher les *plus longs motifs communs*, à toutes les lignes de ce fichier texte, *sans doublons*.

Exemple :

On suppose que le fichier **'C:\génomes.txt'** est composé des lignes suivantes :

G**ATTCGAG**GCTAAA**CTAGCTA**A
G**ATTCGAG**GCT**CTAGCTATTCGA**GGG
CTAGCTATAGG**CTAGCTA**CCATT**ATTCGAG**
CG**CTAGCTA**C**ATTCGAG**GCTAAAC
ATTCGAGAGGGCTAA**CTAGCTA**AGCCA

Le programme affiche les plus longs motifs communs à toutes les lignes de ce fichier :

'ATTCGAG' , 'CTAGCTA'

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ **FIN** ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~