Épreuve d'Informatique – Session 2018 – Filière PSI

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.

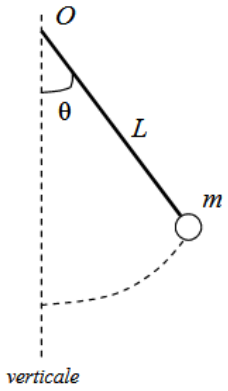
Remarques générales :

- ✓ *L'épreuve se compose de trois 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 : Base de données d'un pendule simple

Un pendule simple est constitué d'un solide de masse **m**, suspendu à une tige rigide de longueur **L** et de masse négligeable devant **m**. Ecarté de sa position initiale d'un angle $\boldsymbol{\theta}$ et lâché sans vitesse initiale, le pendule effectue des oscillations périodiques autour de sa position d'équilibre définie par : $\boldsymbol{\theta = 0}$.



On considère une base de données, dans laquelle on a enregistré les mesures réalisées sur plusieurs pendules simples. Cette base de données est composée de deux tables : la table **Pendules** et la table **Mesures**.

Pendules	
code	(texte)
longueur	(réel)
masse	(réel)
coef_frottement	(réel)

Mesures	
cod_pendule	(texte)
T	(réel)
theta	(réel)

a- Structure de la table **Pendules** :

La table **Pendules** est composée de **4** champs :

Pendules (**code** , **longueur** , **masse** , **coef_frottement**)

- ✓ Le champ **code** contient un code unique pour chaque pendule ;
- ✓ Le champ **longueur** contient la longueur de la tige du pendule, exprimée en *mètre* ;
- ✓ Le champ **masse** contient la masse de l'objet suspendu, exprimée en *kilogramme* ;
- ✓ Le champ **coef_frottement** contient le coefficient du frottement. Si la valeur de ce champ est **0**, le pendule est dit : *sans frottement*.

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

La table **Mesures** est composée de **3** champs :

Mesures (**cod_pendule** , **T , theta**)

- ✓ Le champ **cod_pendule** contient les codes des pendules ;
- ✓ Le champ **T** contient les temps de mesure de l'angle thêta, exprimé en *seconde*.

  NB : Les mesures de l'angle thêta sont prises à un même intervalle de temps, pour chaque pendule.
- ✓ Le champ **theta** contient la mesure de l'angle thêta correspondante à chaque instant de **T**, exprimée en *radian*.

c- Exemples de données dans les deux tables :

Table : Pendules			
code	***longueur***	***masse***	***coef_frottement***
P1	0.5	1.0	0.1
P2	0.4	0.5	0.0
P3	1.0	0.5	0.4
P4	0.5	2.0	0.1
P5	1.0	0.5	0.0
…	…	…	…

Table : Mesures		
cod_pendule	***T***	***theta***
P1	**0**	1.047198
P1	0.005	1.046348
P1	0.010	1.045502
P1	0.015	1.043807
P1	0.020	1.042119
…	…	…
P1	**10**	0.120374
P2	**0**	1.570796
P2	0.02	1.551176
P2	0.04	1.531556
P2	0.06	1.492324
…	…	…
P2	**25**	- 0.387401
P3	**0**	2.356194
P3	0.02	2.350645
P3	0.04	2.345273
P3	…	…

Q.1- Déterminer la *clé primaire* de la table **Pendules**, et la *clé primaire* de la table **Mesures**.

Justifier votre réponse.

Q.2- Convertir la requête suivante, en langage SQL :

$$R = \pi_{T,\ theta} (\sigma_{cod\_pendule = 'P1'} (Mesures))$$

Q.3- Écrire une requête SQL, qui donne pour résultat, **les codes, les masses et les longueurs des pendules avec frottement, dont le code ne contient pas le chiffre 5.**

Q.4- Écrire une requête SQL, qui donne pour résultat, **les codes des pendules et les mesures, exprimées en degré, de l'angle thêta à l'instant 0, triés dans l'ordre décroissant des mesures de l'angle thêta.**

Q.5- Écrire une requête SQL, qui donne pour résultat, **les codes des pendules et le nombre de mesures prises pour chaque pendule, triés dans l'ordre croissant du nombre de mesures.**

Q.6- Pour chaque pendule, les mesures de l'angle thêta ont été prises à un même intervalle de temps, appelé : **pas**.

Écrire une requête SQL, qui donne pour résultat, **les codes des pendules sans frottement, ayant pour pas 10^{-2}.**

Q.7- Écrire une requête SQL, qui **crée la nouvelle table 'Pendules_1'**, ayant la structure suivante :

Pendules_1 (code (texte) , **theta** (réel) , **Tmax** (réel) , **pas** (réel) , **N** (entier)**)**

Q.8- Écrire une requête SQL, qui **copie, dans la table Pendules_1, le code, l'angle thêta initial, le max des T, le pas, et le nombre de mesures prises pour chaque pendule de la table Pendules.**

Exemples de d'enregistrements copiés dans la table **Pendules_1** :

code	*theta*	*Tmax*	*pas*	*N*
P1	1.047198	10	0.005	2000
P2	1.570796	25	0.02	1250
P3	2.356194	30	0.02	1500
…	…	…	…	…

* *

Partie II : Calcul numérique

Résolution numérique d'équation différentielle

La méthode de Nyström

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 :

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

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 y_{n+1} au nœud t_{n+1} fait intervenir la valeur 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 y_{n+1} au nœud t_{n+1} fait intervenir plusieurs valeurs y_n , y_{n-1} , y_{n-2} , … obtenues aux abscisses précédentes. Les principales méthodes sont celles de : *Nyström, Adams-Bashforth, Adams-Moulton, Gear …*

La méthode de Nyström :

La méthode de **Nyström** est une méthode à deux pas, son algorithme est :

> y_0 donné ;
>
> y_1 calculé par la méthode Euler ;
>
> $y_{n+1} = y_{n-1} + 2 * h * f(y_n , t_n)$

Géométriquement : on considère la droite de pente $f(y_n , t_n)$ passant par le point (y_{n-1} , t_{n-1}) parallèle à la tangente passant par (y_n , t_n). La valeur y_{n+1} est l'ordonnée du point de cette droite d'abscisse t_{n+1}.

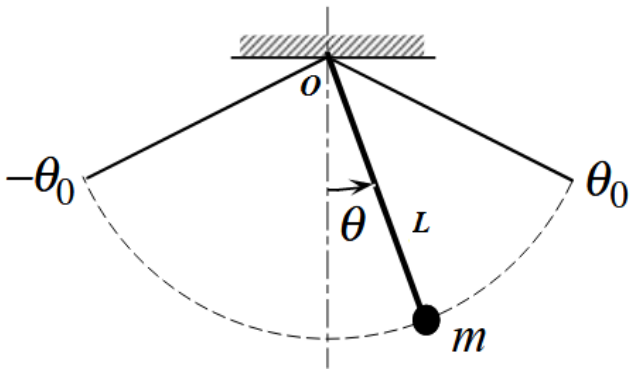
Q. 1 : Écrire une fonction : **nystrom (*f*, a, b, y0, h)**, qui reçoit en paramètres :

- ✓ ***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 de la condition initiale à l'instant **a** (y(a)=y0) ;
- ✓ **h** est le pas de discrétisation.

En utilisant le schéma de *Nyström*, 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 au pendule simple



On considère une masse ***m*** suspendue par une tige rigide de longueur ***L*** et de masse négligeable. On désigne par $\boldsymbol{\theta}$ l'angle entre la verticale passant par le point de suspension et la direction de la tige. On considère que le pendule est également soumis à un frottement fluide de coefficient ***k***.

Si l'objet ***m*** est écarté de façon à ce que l'angle entre la verticale et la tige soit $\boldsymbol{\theta_0}$, et ensuite relâché sans vitesse initiale, alors l'objet ***m*** effectue un mouvement harmonique amorti.

La forme générale de l'équation différentielle modélisant ce mouvement harmonique est :

$$\textbf{(E)} \qquad \boldsymbol{L * \ddot{\theta}(t) + \frac{k}{m*L} * \dot{\theta}(t) + g * sin(\theta(t)) = 0}$$

Les paramètres de cette équation différentielle sont :

- ✓ ***k*** : le coefficient du frottement fluide ;
- ✓ ***g*** : la pesanteur ;
- ✓ ***L*** : la longueur de la tige ;
- ✓ ***m*** : la masse de l'objet suspendu.

NB : *Cette équation n'a pas de solution analytique, sa solution est numérique.*

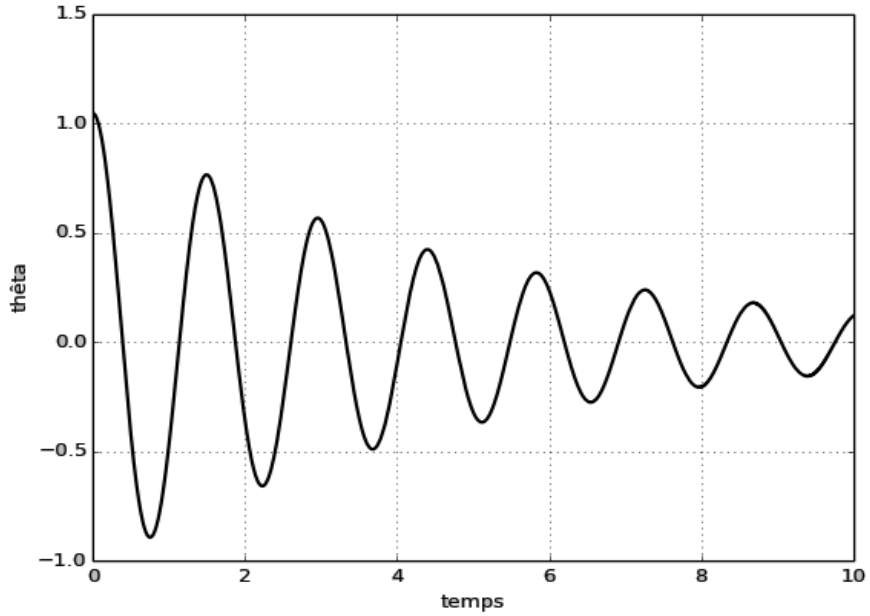
Q. 2 : On suppose que les valeurs des paramètres de l'équation différentielle **(E)**, sont :

$$m = 1\ kg \quad ; \quad g = 9.81\ m/s^2 \quad ; \quad L = 0.50\ m \quad et \quad k = 0.1\ kg/s$$

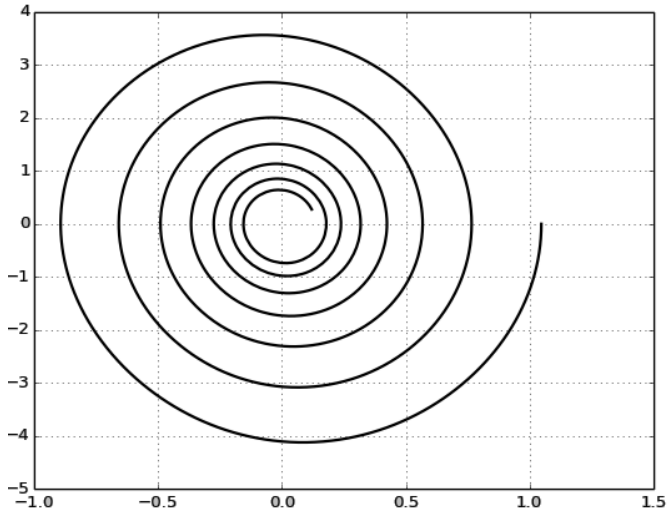
Écrire l'équation différentielle **(E)** sous la forme **z' = F(z, t)**, avec **F(z, t)** une fonction à deux variables **(z, t)** dont **z** est un tableau de longueur **2.**

Q. 3 : On suppose avoir écarté l'objet, de façon à ce que l'angle entre la verticale et la tige, soit de $\pi/3$, avant de le relâcher *sans vitesse initiale*.

3.a- En utilisant la fonction de Nyström, écrire le code python qui, permet de tracer le graphe ci-dessous, qui représente la valeur de l'angle θ, toutes les 10^{-3} **secondes**.



3.b- Écrire le code python permettant de tracer la courbe ci-dessous, qui représente le **portrait de phase du pendule** : les valeurs de $\dot{\theta}$ (axe des ordonnées) en fonction des valeurs de θ (axe des abscisses).



~ ~

Partie III :

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). Les tuples de cette liste sont ordonnés dans l'ordre croissant des positions des suffixes.

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**)]

Recherche séquentielle d'un motif

La **recherche séquentielle**, d'un motif **M** dans une chaîne de caractères **S** non vide, consiste à parcourir la liste **L** des suffixes de **S** (voir Question. 5), en cherchant si le motif **M** est un préfixe du suffixe d'un tuple de la liste **L**. Si oui, la fonction retourne la position de ce suffixe.

*Le parcours, de la liste L, ne doit considérer que les positions possibles pour le motif **M**, c'est-à-dire les positions **p** tels que : **$0 \leq p \leq$ (taille de L – taille de M)**.*

Q. 6 : Écrire une fonction : **recherche_sequentielle (M, L)**, qui, en utilisant le principe de *la recherche séquentielle* dans **L**, renvoie la position du premier tuple, tel que **M** est un préfixe du suffixe de ce tuple, s'il existe ; sinon, la fonction renvoie **None**.

NB : Dans cette fonction, on ne doit pas utiliser la méthode index().

Exemples : La liste des suffixes de la chaîne 'TATCTAGCTA' est :

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

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

Q. 7 : 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_sequentielle** (M, L), dans le pire cas. Et, donner un exemple, de chaînes **M** et **S**, qui atteigne cette complexité.

Tri de la liste des suffixes

Q. 8 : 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. 9 : 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. 10 : 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)**, dans le pire cas. 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 :

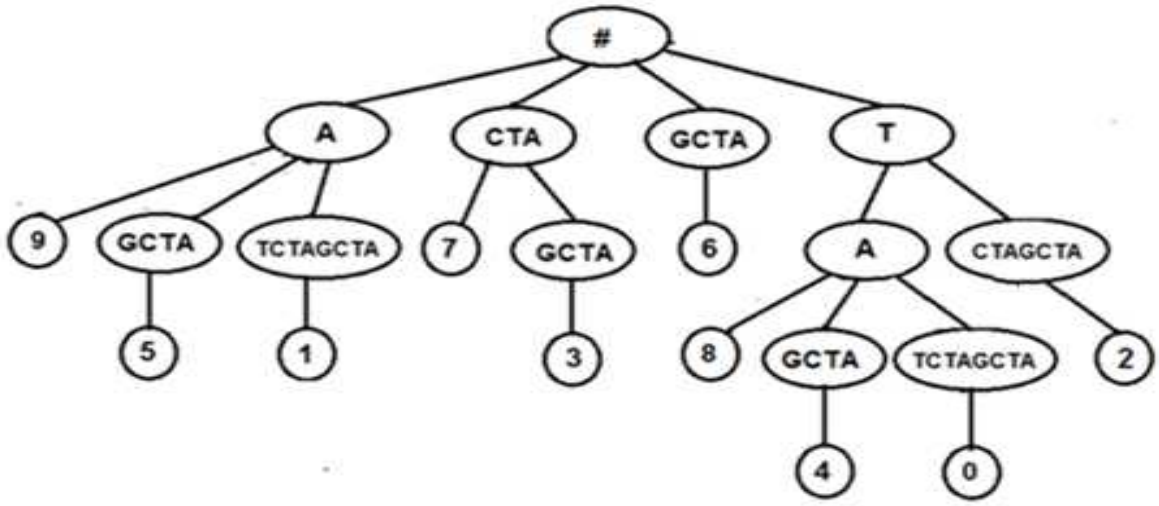
- ✓ La racine de l'arbre est marquée par le caractère '**#**' ;
- ✓ Chaque nœud de l'arbre contient une sous-chaîne de **S**, non vide ;
- ✓ Les premiers caractères, des sous-chaînes fils d'un même nœud, sont différents ;
- ✓ Chaque feuille de l'arbre contient un entier positif qui correspond à la position 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. 11 : 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. 12 : 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 : **[ ]**

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