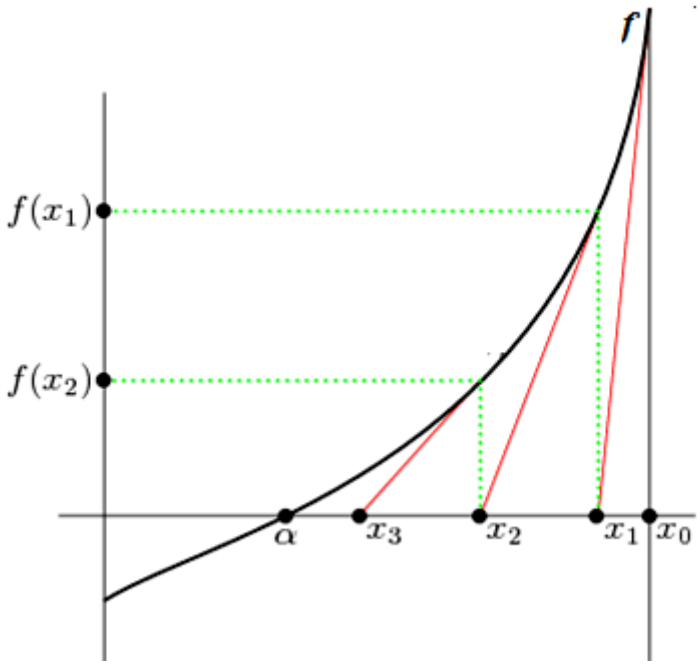
Remarques générales :

- L'épreuve se compose de deux exercices et d'un seul problème qui sont tous indépendants.
- Il faut écrire une fonction en langage **PYTHON** pour chaque question de l'épreuve.
- La précision ainsi que le soin apporté à la rédaction et à la présentation des copies seront des éléments pris en compte dans la notation.
- Les questions non traitées peuvent être admises pour aborder les questions ultérieures.
- Toute fonction peut être décomposée, si nécessaire, en plusieurs fonctions.

Exercice 1 *Calcule de la racine $n^{ième}$ par la méthode de 'Newton'*

De nombreux problèmes d'économie, de mathématiques ou de physique se concluent par la résolution d'une équation $\mathbf{f(x) = 0}$. Bien souvent, il n'est pas possible de résoudre exactement cette équation, et on cherche une valeur approchée de la solution (ou des solutions). **Newton** a proposé une méthode générale pour obtenir une telle approximation. L'idée est de remplacer la courbe représentative de la fonction par sa tangente.



On part d'un point x_0 de l'intervalle de définition de f, et on considère la tangente à la courbe représentative de f en $(x_0 , f(x_0))$. On suppose que cette tangente coupe l'axe des abscisses (c.à.d. $f'(x_0)$ non nul).

Soit x_1 l'abscisse de l'intersection de la tangente avec l'axe des abscisses : $\boldsymbol{x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}}$

Puisque la tangente est proche de la courbe, on peut espérer que x_1 donne une meilleure estimation d'une solution de l'équation $f(x) = 0$ que x_0.

On recommence alors le procédé à partir de x_1, on obtient x_2 : $\boldsymbol{x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}}$

…

Ainsi, on construit par récurrence une suite (x_n) définie par : $\boldsymbol{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}}$

Sous de bonnes hypothèses sur f, assez restrictives, (x_n) converge vers la solution de l'équation $f(x) = 0$, et la convergence est très rapide.

A- Approximation de la dérivée par la moyenne des taux d'accroissement à gauche et à droite :

Si f est une fonction dérivable en x_i, on peut toutefois obtenir une approximation de $f'(x_i)$ par la moyenne des taux d'accroissement à gauche et à droite de x_i :

Pour un réel $h > 0$ (très proche de 0) :

$$f'(x_i) \approx \frac{f(x_i + h) - f(x_i - h)}{2h}$$

Q.1- Montrer que : $x_{n+1} = x_n - 2h\frac{f(x_n)}{f(x_n+h) - f(x_n-h)}$

On a $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ **et** $f'(x_i) \approx \frac{f(x_i+h)-f(x_i-h)}{2h}$ **Donc** $x_{n+1} = x_n - 2h\frac{f(x_n)}{f(x_n+h) - f(x_n-h)}$

Q.2- Écrire une fonction : **newton (*f*, x0, eps, h)** qui reçoit en paramètres :

- La fonction ***f*** ;
- Le premier terme x_0 de la suite (x_n) ;
- Le réel ***h*** strictement positif très proche de **0** ;
- Le réel ***eps*** strictement positif qui représente la précision.

```
def newton ( f, x0, eps, h) :
   while abs((-2*h*f(x0))/(f(x0+h)-f(x0-h))) >eps:
      x0=x0+(-2*h*f(x0))/(f(x0+h)-f(x0-h))
   return x0
```

La fonction renvoie le premier terme x_k de la suite (x_n) tel que : $|x_k - x_{k-1}| \leq eps$

B- Calcul de la racine n-ème positive d'un réel positif

Pour calculer la racine n-ème positive d'un réel positif ***a***, il suffit de trouver la solution positive de l'équation : $x^n - a = 0$, par la méthode de Newton.

Dans ce qui suit, on suppose que ***a*** et ***n*** sont deux variables globales déclarées et initialisées :
a est un réel positif, et ***n*** un entier strictement positif.

Q. 3- Définir la fonction **g (x)**, qui reçoit en paramètre un réel positif ***x*** et qui renvoie : $x^n - a$.

```
def g(x):
   return x**n-a
```

Q. 4- On suppose que les modules suivants sont importés :

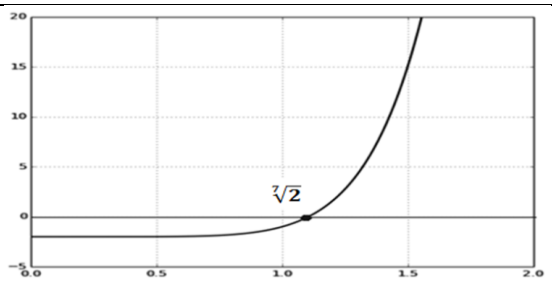
```
import numpy as np
import matplotlib.pyplot as plt
import math as m
```

Écrire le code du programme Python qui permet de tracer la courbe de la fonction ***g***, dans l'intervalle **[0 , a]**.
Le nombre de points générés dans la courbe est : **500**

Exemple :

Pour **a=2.0** et **n=7**, on obtient la courbe suivante :

```
a,n=2,7
X=np.linspace(0,a,500)
Y=g(X)
plt.plot(X,Y)
plt.title("Représentation graphique de la fonction : g(x)")
plt.grid()
plt.show()
```



Représentation graphique de la fonction :
g(x) = x^7 - 2

Q. 5- Écrire le code du programme Python qui utilise la fonction **newton ()**, et qui affiche la valeur approximative de la racine n-ème positive de ***a*** : $\sqrt[n]{a}$, avec la précision : 10^{-12} et $h= 0.00001$.

Exemple :

Pour **n=7** et **a=2.0**, le programme affiche la racine **7ème** positive de **2** : **1.1040895136738123**

```
>>> x0,eps,h=2,1e-12,0.00001
>>> newton ( g, x0, eps, h)
1.1040895136740212
```

Exercice 2 Décomposition en facteurs premiers d'un entier n ≥ 2.

Dans cet exercice, on se propose d'écrire un algorithme pour **décomposer un entier en produit de nombres premiers.**

Pour une nombre premier donné p, on définit la valuation p-adique noté $v_p(n)$ d'un entier non nul n par les règles ci-dessous :

- Si p divise n, on note $v_p(n)$ le plus grand entier k tel que p^k divise n.
- Si p ne divise pas n, on pose $v_p(n) = 0$.

Q1. On considère le code de la fonction **estPremier(n)** qui prend en argument un entier naturel non nul n et qui renvoie le booléen **True** si n est premier et le booléen **False** sinon.

```
1. def estPremier(n):
2.     for i in range(2,n):
3.         if n%i==0:
4.             return False
5.     return True
```

Q1.1. Donner le nombre de tests effectués par la fonction estPremier(n) dans le pire des cas.

(n-2) tests

Q1.2. Déduire la complexité de la fonction estPremier(n) en fonction de n dans le pire des cas.

C(n)= (n-2) * (2+2)=4n-4 ➔ O(n) complxité linéaire

Q2. Afin d'améliorer la complexité de la fonction **estPremier(n)**, on pourra utiliser le critère suivant : un entier n ≥ 2 qui n'est divisible par aucun entier d ≥ 2 tel que $d^2 \leq n$, est premier.

Q2.1. Compléter le code la fonction **estPremier_am(n),** cette fonction doit prendre en considération le critère mentionné ci-dessus.

```
1. def estPremier_am(n):
2.         for i in range(2,int(n**(1/2))+1):
3.             if n%i==0:
4.                     return False
5.         return True
```

Q2.2. Quelle est la complexité de la fonction **estPremier_am(n)** en fonction de n dans le pire des cas. Justifier votre réponse.

C(n)=E($\sqrt{n}-2$) $*$ (2 + 2) < cte n, cte désigne une constante réelle positive ➔ O(n)

Q3. Ecrire une fonction **liste_premiers(n)** qui prend en argument un entier naturel non nul n et renvoie la liste des nombres premiers inférieurs ou égaux à n.

```
def liste_premiers(n):
    L=[]
    for k in range(2,n+1):
        if estPremier_am(k):
            L=L+[k]
    return L
```

Exemple d'exécution :

>>>liste_premiers(11)
[2, 3, 5,7,11].

Par exemple, pour calculer la valuation 2-adique de 40, on peut utiliser la méthode suivante :

- 40 est divisible par 2 et le quotient vaut 20
- 20 est divisible par 2 et le quotient vaut 10
- 10 est divisible par 2 et le quotient vaut 5
- 5 n'est pas divisible par 2.

La valuation 2-adique de 40 vaut donc 3.

Q4. Écrire une fonction iterative **valuation_p_adique_it(n, p)** qui implémente cet algorithme. Elle prend en arguments un entier naturel n non nul et un nombre premier p et renvoie la valuation p-adique de n.
Par exemple, puisque $40 = 2^3 \times 5$, valuation_p_adique(40, 2) renvoie 3, valuation_p_adique(40, 5) renvoie 1 et valuation_p_adique(40, 7) renvoie 0.

```
def valuation_p_adique_it(n, p):
    if n%p!=0:
        return 0
    else:
        k=1
        while n%(p**k)==0:
            k=k+1
        return k-1
```

Q5. Écrire une deuxième fonction cette fois-ci récursive **val_p_adique_re(n,p)** qui renvoie la valuation p-adique de n.

```
def val_p_adique_re(n,p):
    if n%p!=0:
        return 0
    return 1+val_p_adique_re(n/p,p)
```

Q6. En déduire une fonction **decomposition_facteurs_premiers(n)** qui calcule la décomposition en facteurs premiers d'un entier $n \geq 2$.
Cette fonction doit renvoyer la liste des couples $(p, v_p(n))$ pour tous les nombres premiers p qui divisent n.

```
def decomposition_facteurs_premiers(n):
    L=liste_premiers(n)
    DFP=[]
    for e in L:
        if n%e==0:
            DFP=DFP+[[e,valuation_p_adique_it(n,e)]]
    return DFP
```

Exemple d'exécution :
>>> decomposition_facteurs_premiers(40)
[[2, 3], [5, 1]]

Problème 1. 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.

Question 1. Quel est le nombre minimum de bits nécessaires pour coder quatre éléments ? deux bits

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

'A'=00, 'C'=01, 'G'=10 et 'T'=11

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 'TATCTAGCTA' ;
- La chaîne 'TATCTA' est un préfixe de 'TATCTAGCTA' ;
- La chaîne 'TATCTAGCTA' est un préfixe de 'TATCTAGCTA' ;
- La chaîne 'T' est un préfixe de 'TATCTAGCTA' ;
- La chaîne 'CTAGC' n'est pas un préfixe de S.

Question 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.

```
def prefixe (M,S):
    for i in range(1,len(S)):
        if M==S[:i]:
            return True
    return False
```

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

Question 4. Écrire une fonction **liste_suffixes (S),** qui reçoit en paramètre une chaîne de caractères S non vide, et qui renvoie une **liste de listes**. Chaque **liste** 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]]

```
def liste_suffixes (S):
    LS=[]
    for i in range(0,len(S)):
        LS=LS+[[S[i:],i]]
    return LS
```

Tri de la liste des suffixes

On se donne la fonction tri suivante, écrite en Python :

```
def tri(L):
    n=len(L)
    for i in range(n):
        for j in range(n-1):
            if L[j]>L[j+1]:
                L[j],L[j+1]=L[j+1],L[j]
```

Question 5. Lors de l'appel tri(L) lorsque L est la liste ['BAC','AB','DA','AC'], donner le contenu de la liste L à la fin de chaque itération de **la première boucle for** (for i in range(n)).

```
>>> L=['BAC','AB','DA','AC']
>>> tri(L)
['AB', 'BAC', 'AC', 'DA']
['AB', 'AC', 'BAC', 'DA']
['AB', 'AC', 'BAC', 'DA']
['AB', 'AC', 'BAC', 'DA']
```

Question 6. Donnez la complexité de la fonction tri(L) en fonction de n (n=len(L)) dans le meilleur des cas et le pire des cas.

le meilleur cas (lorsque L est triée d'une manière croissante) : C(n)= 2+6(n-2)+ 2+6(n-3)+…+ 2+6x2+2+6x1+1 donc il s'agit d'une complexité quadratique $O(n^2)$

le meilleur pire (lorsque L est triée d'une manière décroissante) : C(n)= C(n)= 2+9(n-2)+ 2+9(n-3)+…+ 2+9x2+2+9x1+1 donc il s'agit d'une complexité quadratique $O(n^2)$

		Cas meilleur	Cas pire
	def tri(L):		
1	n=len(L)	1	1
2	for i in range(n):	2	2
3	for j in range(n-1):	2	2
4	if L[j]>L[j+1]:	4	4
5	L[j],L[j+1]=L[j+1],L[j]	0	9

Question 7. 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 4. 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().

```
def tri_liste(L):
    n=len(L)
    for i in range(n):
        for j in range(n-1):
            if L[j][0]>L[j+1][0]:
                L[j],L[j+1]=L[j+1],L[j]
```

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 **listes** composées des suffixes d'une chaîne de caractères S non vide, et de leurs positions dans S.
Question 8. Écrire une fonction récursive **recherche(M,L)**, qui reçoit en paramètre une le motif M et une liste des suffixes L. Si M est un préfixe d'un suffixe dans une **liste** 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.

```
def recherche(M,L):
    if L==[]:
        return None
    if prefixe (M,L[0][0]):
        return L[0][1]
    else:
        return recherche(M,L[1:])
```

Exemple :
La liste des suffixes de 'TATCTAGCTA', est :
L = [['TATCTAGCTA', 0], ['ATCTAGCTA', 1], ['TCTAGCTA', 2], ['CTAGCTA', 3], ['TAGCTA', 4], ['AGCTA', 5], ['GCTA', 6], ['CTA', 7], ['TA', 8], ['A', 9]]

- recherche('CTA', L) renvoie la position 3 ; 'CTA' est préfixe de 'CTAGCTA'
- recherche('TA', L) renvoie la position 0 ; 'TA' est préfixe de 'TATCTAGCTA'
- recherche('CAGT', L) renvoie None. 'CAGT' n'est préfixe d'aucun suffixe

Question 9. Donnez la complexité de la fonction recherche(M,L) en fonction de n (n=len(L)) dans de pire des cas. Justifier votre réponse

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'

Question 10. 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.

```
def matrice(P,Q):
    M=len(P)*[len(Q)*[0]]
    for j in range(0,len(Q)):
        if P[0]==Q[j]:
            M[0][j]=1
    for i in range(1,len(P)):
        if P[i]==Q[0]:
            M[0][j]=1
    for i in range(1,len(P)):
        for j in range(1,len(Q)):
            if P[i]==Q[j]:
                M[i][j]=M[i-1][j-1]+1
    return M
```

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], [...] , ...]

Question 11. 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.

```
def plus_long_mc (P, M):
    (i,j)=max(M)
    L=[(i,j)]
    while i>0 and j>0 and M[i][j]!=0:
        L=L+[(i-1,j-1)]
        i=i-1
        j=j-1
    return [P[L[k][0]] for k in range(len(L)-1,-1,-1)]
```

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']