Autour des nombres premiers

Préambule

Chiffrer les données est nécessaire pour assurer la confidentialité lors d'échanges d'informations sensibles. Dans ce domaine, les nombres premiers servent de base au principe de clés publique et privée qui permettent, au travers d'algorithmes, d'échanger des messages chiffrés. La sécurité de cette méthode de chiffrement repose sur l'existence d'opérations mathématiques peu coûteuses en temps d'exécution mais dont l'inversion (c'est-à-dire la détermination des opérandes de départ à partir du résultat) prend un temps exorbitant. On appelle ces opérations « fonctions à sens unique ». Une telle opération est, par exemple, la multiplication de grands nombres premiers. Il est aisé de calculer leur produit. Par contre, connaissant uniquement ce produit, il est très difficile de déduire les deux facteurs premiers.

Le sujet étudie différentes questions sur les nombres premiers.

Les programmes demandés sont à rédiger en langage `Python` 3. Si toutefois le candidat utilise une version antérieure de Python, il doit le préciser. Il n'est pas nécessaire d'avoir réussi à écrire le code d'une fonction pour pouvoir s'en servir dans une autre question. Les questions portant sur les bases de données sont à traiter en langage `SQL`.

Définitions, rappels et notations

- Un nombre premier est un entier naturel qui admet exactement deux diviseurs : 1 et lui-même. Ainsi 1 n'est pas considéré comme premier.
- Un flottant est la représentation d'un nombre réel en mémoire.
- Quand une fonction `Python` est définie comme prenant un « nombre » en paramètre cela signifie que ce paramètre pourra être indifféremment un flottant ou un entier.
- On note $\lfloor x \rfloor$ la partie entière de x.
- `abs(x)` renvoie la valeur absolue de `x`. La valeur renvoyée est du même type de données que celle en argument.
- `int(x)` convertit vers un entier. Lorsque x est un flottant positif ou nul, elle renvoie la partie entière de x, c'est-à-dire l'entier n tel que $n \leqslant x < n+1$.
- `round(x)` renvoie la valeur de l'entier le plus proche de `x`. Si deux entiers sont équidistants, l'arrondi se fait vers la valeur paire.
- `floor(x)` renvoie la valeur du plus grand entier inférieur ou égal à `x`.
- `ceil(x)` renvoie la valeur du plus petit entier supérieur ou égal à `x`.
- `log(x)` renvoie sous forme de flottant la valeur du logarithme népérien de `x` (supposé strictement positif).
- `log(x,n)` renvoie sous forme de flottant la valeur du logarithme de `x` en base `n`.
- La fonction `time()` du module `time` renvoie un flottant représentant le nombre de secondes depuis le 01/01/1970 avec une résolution de 10^{-7} seconde (horloge de l'ordinateur).
- L'opérateur usuel de division `/` renvoie toujours un flottant, même si les deux opérandes sont des multiples l'un de l'autre.
- L'infini $+\infty$ en `Python` s'écrit `float("inf")`.
- En `Python` 3, on peut utiliser des entiers illimités de plus de 32 bits avec le type `long`.

Partie I. Préliminaires

❏ **Q1** – Dans un programme `Python` on souhaite pouvoir faire appel aux fonctions `log`, `sqrt`, `floor` et `ceil` du module `math` (`round` est disponible par défaut). Écrire des instructions permettant d'avoir accès à ces fonctions et d'afficher le logarithme népérien de 0.5.

❏ **Q2** – Écrire une fonction `sont_proches(x, y)` qui renvoie `True` si la condition suivante est remplie et `False` sinon

$$|x - y| \leqslant atol + |y| \times rtol$$

où *atol* et *rtol* sont deux constantes, à définir dans le corps de la fonction, valant respectivement 10^{-5} et 10^{-8}. Les paramètres `x` et `y` sont des nombres quelconques.

❏ **Q3** – On donne la fonction `mystere` ci-dessous. Que renvoie `mystere(1001,10)` ? Le paramètre `x` est un nombre strictement positif et `b` un entier naturel non nul.

```python
def mystere(x,b):
    if x < b:
        return 0
    else:
        return 1 + mystere(x / b, b)
```

❏ **Q4** – Exprimer ce que renvoie `mystere` en fonction de la partie entière d'une fonction usuelle.

❏ **Q5** – On donne le code suivant :

```python
pas = 1e-5

x2 = 0
for i in range(100000):
    x1 = (i + 1) * pas
    x2 = x2 + pas

print("x1:", x1)
print("x2:", x2)
```

L'exécution de ce code produit le résultat :

```
x1: 1.0
x2: 0.9999999999980838
```

Commenter.

Partie II. Génération de nombres premiers

II.a Approche systématique

Le crible d'Ératosthène est un algorithme qui permet de déterminer la liste des nombres premiers appartenant à l'intervalle $[\![1,n]\!]$. Son pseudo-code s'écrit comme suit :

Données : N, entier supérieur ou égal à 1
Résultat : *liste_bool*, liste de booléens
début
  liste_bool $\longleftarrow$ liste de N booléens initialisés à Vrai;
  Marquer comme Faux le premier élément de *liste_bool*;
  pour *entier* $i \leftarrow 2$ **à** $\lfloor\sqrt{N}\rfloor$ **faire**
    si *i n'est pas marqué comme Faux dans liste_bool* **alors**
      Marquer comme Faux tous les multiples de i différents de i dans *liste_bool* ;
    fin
  fin
  retourner *liste_bool*
fin

Algorithme 1 : Crible d'Ératosthène

À la fin de l'exécution, si un élément de `liste_bool` vaut Vrai alors le nombre codé par l'indice considéré est premier. Par exemple pour `N=4` une implémentation `Python` du crible renvoie `[False True True False]`.

❑ **Q6** – Sachant que le langage `Python` traite les listes de booléens comme une liste d'éléments de 32 bits, quel est (approximativement) la valeur maximale de `N` pour laquelle `liste_bool` est stockable dans une mémoire vive de 4 Go ?

❑ **Q7** – Quel facteur peut-on gagner sur la valeur maximale de `N` en utilisant une bibliothèque permettant de coder les booléens non pas sur 32 bits mais dans le plus petit espace mémoire possible pour ce type de données (on demande de le préciser) ?

❑ **Q8** – Écrire la fonction `erato_iter(N)` qui implémente l'algorithme 1 pour un paramètre `N` qui est un entier supérieur ou égal à 1.

❑ **Q9** – Quelle est la complexité algorithmique du crible d'Ératosthène en fonction de N ? On admettra que :

$$\sum_{p<N,p\ premier} \frac{1}{p} \simeq \ln(\ln(N)) \tag{1}$$

La réponse devra être justifiée.

❑ **Q10** – Quand on traite des nombres entiers il est intéressant d'exprimer la complexité d'un algorithme non pas en fonction de la valeur N du nombre traité mais de son nombre de chiffres n. Donner une approximation du résultat de la question précédente en fonction de n en précisant la base choisie.

II.b Génération rapide de nombres premiers

L'approche systématique qui précède est inefficace car elle revient à attendre d'avoir généré la liste de tous les nombres premiers inférieurs à une certaine valeur pour en choisir ensuite quelques uns au hasard. Une meilleure idée est d'utiliser des tests probabilistes de primalité. Ces tests ne garantissent pas vraiment qu'un nombre est premier. Cependant, au sens probabiliste, si un nombre

réussit un de ces tests alors la probabilité qu'il ne soit pas premier est prouvée être inférieure à un seuil calculable.
En suivant cette idée, une nouvelle approche est la suivante :

1. générer un entier pseudo-aléatoire (voir ci-dessous)
2. vérifier si cet entier a de fortes chances d'être premier
3. recommencer tant que le résultat n'est pas satisfaisant.

Pour générer un entier pseudo-aléatoire A on se base sur un certain nombre d'itérations de l'algorithme Blum Blum Shub, décrit comme suit. On initialise A à zéro au début de l'algorithme et pour chaque itération ($i \geq 1$) on calcule :

$$x_i = \text{reste de la division euclidienne de } x_{i-1}^2 \text{ par } M \tag{2}$$

où M est le produit de deux nombres premiers quelconques et x_0 une valeur initiale nommée « graine » choisie aléatoirement. On utilise ici l'horloge de l'ordinateur comme source pour x_0.
Puis, pour chaque x_i, s'il est impair, on additionne 2^i à A.

❏ **Q11** – On répète (2) pour i parcourant $[\![1,N-1]\!]$, quelle sera la valeur de A si x_i est impair à chaque itération ?

❏ **Q12** – Compléter (avec le nombre de lignes que vous jugerez nécessaire) la fonction `bbs(N)` donnée ci-dessous qui réalise ces itérations. La graine est un entier représentant la fraction de secondes du temps courant, par exemple 1528287738.7931523 donne la graine 7931523. Le paramètre `N` est un entier non nul.

```
...
...
def bbs(N):
    p1 = 24375763
    p2 = 28972763
    M = p1 * p2
    # calculer la graine
    ... (à compléter)
    A = 0
    for i in range(N):
        if ... (à compléter) # si xi est impair
            A = A + 2**i
        # calculer le nouvel xi
        xi = ... (à compléter)
    return(A)
```

Le test probabiliste de primalité le plus simple est le test de primalité de Fermat. Ce test utilise la contraposée du petit théorème de Fermat qu'on peut évoquer comme suit : si $a \in [\![2, p-1]\!]$ est premier et que le reste de la division euclidienne de a^{p-1} par p vaut 1, alors il y a de "fortes" chances pour que p soit premier.

❏ **Q13** – En combinant les résultats du test de primalité de Fermat pour $a = 2$, $a = 3$, $a = 5$ et $a = 7$, écrire une fonction `premier_rapide(n_max)` qui renvoie un nombre aléatoire inférieur strictement à *n_max* qui a de fortes chances d'être premier. Le paramètre `n_max` est un entier supérieur à 12.

❏ **Q14** – On souhaite caractériser le taux d'erreurs de `premier_rapide`.
Écrire une fonction `stats_bbs_fermat(N, nb)` qui contrôle pour `nb` nombres, inférieurs ou égaux à `N`, générés par `premier_rapide`, qu'ils sont réellement premiers. Cette fonction renvoie le taux relatif d'erreur ainsi que la liste des faux nombres premiers trouvés. Les paramètres `N` et `nb` sont des entiers strictement positifs.

Partie III. Compter les nombres premiers

La question de la répartition des nombres premiers a été étudiée par de nombreux mathématiciens, dont Euclide, Riemann, Gauss et Legendre. On étudie dans cette partie les propriétés de la fonction $\pi(n)$, qui renvoie le nombre de nombres premiers appartenant à $[\![1,n]\!]$.

III.a Calcul de $\pi(n)$ via un crible

❏ **Q15** – Écrire une fonction `Pi(N)` qui calcule la valeur exacte de $\pi(n)$ pour tout entier n de $[\![1,N]\!]$. Les nombres premiers sont déduits de la liste `liste_bool` renvoyée par la fonction `erato_iter` de la question 8. On demande que `Pi(N)` renvoie son résultat sous la forme d'une liste de `[n,` $\pi(n)$`]`. Par exemple `Pi(4)` renvoie la liste [[1, 0], [2, 1], [3, 2], [4, 2]].
Un seul appel à `erato_iter` est autorisé et on exige une fonction dont la complexité, en dehors de cet appel, est linéaire en fonction de `N`. Le paramètre `N` est un entier supérieur à 1.

Il a été prouvé que $\frac{n}{\ln(n)-1} < \pi(n)$ pour tout $n \geqslant 5393$. On souhaite vérifier cette inégalité en se basant sur la fonction `Pi(N)` écrite en Question 15.

❏ **Q16** – Écrire une fonction `verif_Pi(N)` qui renvoie `True` si l'inégalité est vérifiée jusqu'à `N` inclus, `False` sinon. Le paramètre `N` est un entier supposé supérieur ou égal à 5393.

III.b Calcul d'une valeur approchée de $\pi(n)$

Le calcul de $\pi(n)$ dépend de la capacité à calculer de manière exhaustive tous les nombres premiers de $[\![1,N]\!]$, or le temps nécessaire à ce calcul devient rapidement très grand lorsque N augmente. Il existe en revanche diverses méthodes pour calculer une valeur approchée de $\pi(n)$. Une méthode utilise la fonction logarithme intégral li, dont une représentation graphique est fournie en figure 1, et qui est définie comme :

$$\begin{aligned} \mathrm{li} : \mathbb{R}^+ \setminus \{1\} &\to \mathbb{R} \\ x &\mapsto \int_0^x \frac{\mathrm{dt}}{\ln(t)} \end{aligned} \tag{3}$$

Si $x > 1$, l'intégrale impropre doit être interprétée comme sa valeur principale de Cauchy qui est définie comme :

$$\mathrm{li}(x) = \lim_{\varepsilon \to 0^+} \left(\int_0^{1-\varepsilon} \frac{\mathrm{dt}}{\ln(t)} + \int_{1+\varepsilon}^x \frac{\mathrm{dt}}{\ln(t)} \right) \tag{4}$$

L'intérêt de li pour compter les nombres premiers vient de la propriété suivante :

$$\lim_{x \to \infty} \frac{\pi(\lfloor x \rfloor)}{\mathrm{li}(x)} = 1 \tag{5}$$

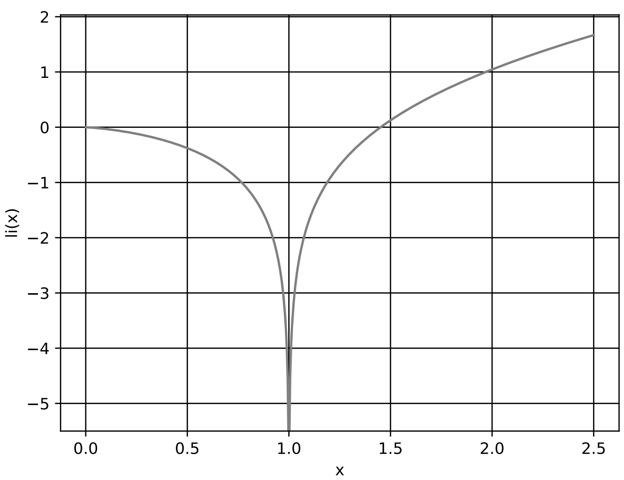

FIGURE 1 – Allure de li sur [0,2.5]

On souhaite développer un programme permettant de calculer une valeur approchée de li. On compare ensuite les résultats obtenus à une implémentation de référence qui est nommée `ref_li`, réputée très précise.

Estimation de li par quadrature numérique

On choisit d'utiliser la méthode des rectangles à droite. On appelle *pas* la base des rectangles. La figure 4 illustre cette méthode appliquée au calcul de la valeur principale définie équation (4).
Par souci de simplification, on suppose que *pas* est choisi de manière à ce que 1 et x soient multiples de *pas* et on utilise $\varepsilon = pas$ dans l'équation (4).

❏ **Q17** – La fonction qui est évaluée sur l'intervalle d'intégration est supposée avoir une complexité constante quelles que soient ses valeurs d'entrée. Quelle est la complexité en temps de la méthode des rectangles à droite ? On prendra soin d'expliciter en fonction de quelle variable d'entrée cette complexité est exprimée.

❏ **Q18** – Dans les mêmes conditions d'évaluation, quelle est la complexité en temps de la méthode des rectangles centrés ? Donner aussi celle de la méthode des trapèzes.

❏ **Q19** – Écrire une fonction `inv_ln_rect_d(a, b, pas)` qui calcule par la méthode des rectangles à droite une valeur approchée de $\int_a^b \frac{\mathrm{dt}}{\ln(t)}$ avec une incrément valant `pas`. On suppose dans cette question que $a < b$ et que 1 n'appartient pas à l'intervalle $[a, b]$ de sorte que la fonction intégrée est définie et continue sur $[a, b]$.
On considère que le réel $b - a$ est un multiple du réel *pas*.
Les paramètres `a`, `b` et `pas` sont des flottants.

❏ **Q20** – Écrire une fonction `li_d(x, pas)` qui calcule une valeur approchée de $\mathrm{li}(x)$ avec la méthode des rectangles à droite en se basant sur `inv_ln_rect_d`. Si $x = 1$ la fonction renvoie $-\infty$. On rappelle qu'on suppose que *pas* est choisi de manière à ce que 1 et x soient multiples de *pas* et qu'on utilise $\varepsilon = pas$ dans l'équation (4). Les paramètres `x` et `pas` sont des flottants.

Analyse des résultats de `li_d`

Après avoir testé `li_d` on obtient plusieurs résultats surprenants.

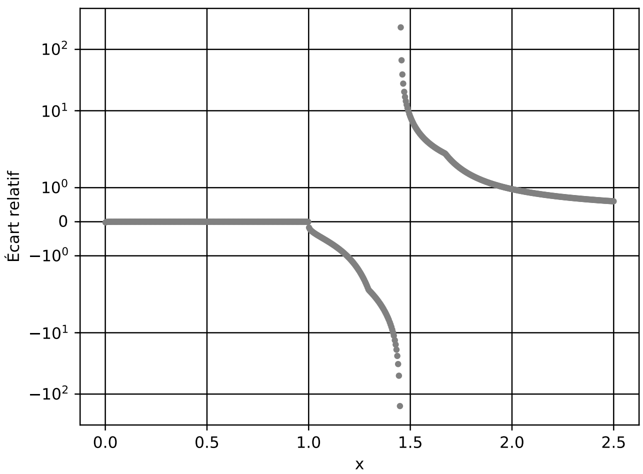

FIGURE 2 – Écart relatif entre `li_d` ($pas = 10^{-4}$) et l'implémentation de référence `ref_li`.

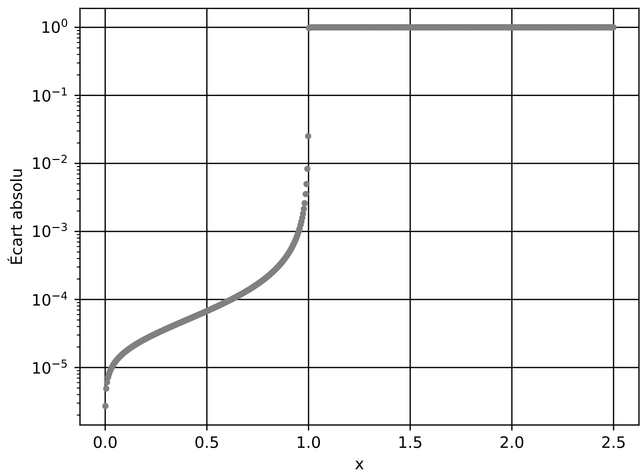

FIGURE 3 – Écart absolu entre `li_d` ($pas = 10^{-4}$) et l'implémentation de référence `ref_li`.

❏ **Q21** – Expliquer le comportement de l'écart relatif entre `li_d` et `ref_li`, illustré figure 2 au voisinage de $x \simeq 1.4$.

❑ **Q22** – On constate un écart absolu important entre `li_d` et `ref_li` au delà de $x = 1$, illustré figure 3. Expliquer succinctement d'où vient ce phénomène. On ne demande pas une démonstration mathématique rigoureuse.
Pour répondre à cette question on pourra remarquer qu'au premier ordre $\frac{1}{ln(1+\varepsilon)} \simeq -\frac{1}{ln(1-\varepsilon)}$ quand $\varepsilon \to 0$ et s'interroger sur la valeur que devrait avoir l'intégrale impropre de $\frac{1}{\ln(x)}$ sur un intervalle $[1-\varepsilon, 1+\varepsilon]$ avec $\varepsilon \ll 1$. Une analyse géométrique de la figure 4 peut aussi s'avérer utile.

❑ **Q23** – Proposer, en justifiant votre choix, une ou des modifications de l'algorithme utilisé afin d'éliminer le problème constaté sur l'écart absolu. Il n'est pas demandé d'écrire le code mettant en œuvre ces propositions.

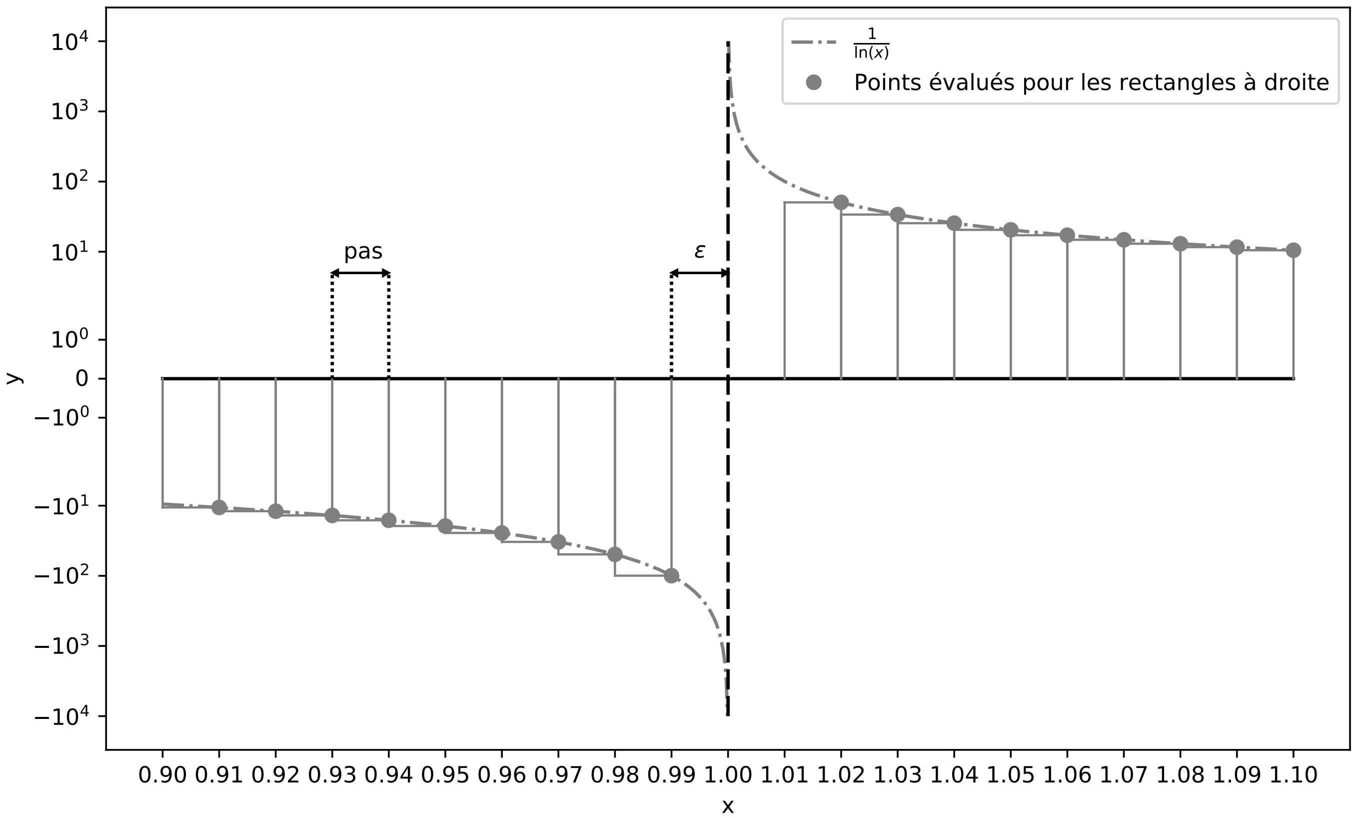

FIGURE 4 – Rectangles utilisés par la méthode des rectangles à droite au voisinage de 1 afin de calculer la valeur principale de Cauchy introduite dans l'équation (4). Les paramètres sont $pas = \varepsilon = 10^{-2}$

Estimation de li via Ei

L'approche par quadrature numérique n'est pas satisfaisante. Non seulement elle rend le temps d'exécution de `li_d` prohibitif quand x augmente mais de plus l'utilisateur doit choisir un pas sans règle claire à appliquer pour garantir une précision donnée. La fonction exponentielle intégrale Ei permet de pallier ce problème.

$$\begin{aligned} \mathrm{Ei} : \mathbb{R}^* &\to \mathbb{R} \\ x &\mapsto \int_{-\infty}^{x} \frac{e^t}{t}\mathrm{dt} \end{aligned} \tag{6}$$

Pour le cas $x > 0$ on utilise la valeur principale de Cauchy telle que vue pour li.
Le lien entre li et Ei est :

$$\mathrm{li}(x) = \mathrm{Ei}(\ln(x)) \tag{7}$$

Afin d'évaluer numériquement la valeur de Ei en un point on se base sur son développement (dit en série de Puiseux) sur $\mathbb{R}^{+*}$:

$$\mathrm{Ei}(x) = \gamma + \ln(x) + \sum_{k=1}^{\infty} \frac{x^k}{k \times k!} \tag{8}$$

Avec $\gamma \simeq 0.577215664901$ la constante d'Euler-Mascheroni.
Comme l'évaluation de la somme jusqu'à l'infini est impossible on utilise en pratique la somme suivante :

$$\mathrm{Ei}_n(x) = \gamma + \ln(x) + \sum_{k=1}^{n} \frac{x^k}{k \times k!} \tag{9}$$

Le choix de n se fait en comparant Ei_{n-1} à Ei_n jusqu'à ce qu'ils soient considérés comme suffisament proches.
L'évaluation via un ordinateur de ce développement est numériquement stable jusqu'à $x = 40$. Au delà les résultats sont entachés d'erreurs de calcul et d'autres méthodes doivent être utilisées.

❏ **Q24** – Écrire une fonction `li_dev(x)` qui calcule $\mathrm{li}(x)$ en se basant sur Ei_n et la fonction `sont_proches` de la question 2 (on pourra utiliser la fonction associée même si la question n'a pas été traitée). `li_dev` doit renvoyer `False` si :

- Ei_{n-1} et Ei_n ne peuvent pas être considérés comme proches au bout de `MAXIT` itérations.
- la valeur de `x` ne permet pas d'aboutir à un résultat.

Prendre `MAXIT = 100` se révèle largement suffisant à l'usage.
On demande à ce que la complexité dans le pire des cas soit $O($`MAXIT`$)$. Le paramètre `x` est un flottant quelconque.

Partie IV. Analyse de performance de code

Au cours du développement des fonctions nécessaires à la manipulation des nombres premiers on s'aperçoit que le choix des algorithmes pour évaluer chaque fonction est primordial pour garantir des performances acceptables. On souhaite donc mener des tests à grande échelle pour évaluer les performances réelles du code qui a été développé. Pour ce faire on effectue un grand nombre de tests sur une multitude d'ordinateurs. Les données sont ensuite centralisées dans une base de données composée de deux tables.
La première table est `ordinateurs` et permet de stocker des informations sur les ordinateurs utilisés pour les tests. Ses attributs sont :

- `nom TEXT`, clé primaire, le nom de l'ordinateur.
- `gflops INTEGER` la puissance de l'ordinateur en milliards d'opérations flottantes par seconde.
- `ram INTEGER` la quantité de mémoire vive de l'ordinateur en Go.

Exemple du contenu de cette table :

```
nom                 gflops      ram
----------          ----------  ----------
nyarlathotep114     69          32
nyarlathotep119     137         32
...
shubniggurath42     133         16
azathoth137         85          8
```

La seconde table est `fonctions` et stocke les informations sur les tests effectués pour différentes fonctions en cours de développement. Ses attributs sont :

- `id INTEGER` l'identifiant du test effectué.
- `nom TEXT` le nom de la fonction testée (par exemple li, Ei, etc).
- `algorithme TEXT` le nom de l'algorithme qui permet le calcul de la fonction testée (par exemple BBS si on teste une fonction de génération de nombres aléatoires).
- `teste_sur TEXT` le nom du PC sur lequel le test a été effectué.
- `temps_exec INTEGER` le temps d'exécution du test en millisecondes.

Exemple du contenu de cette table :

```
id          nom         algorithme  teste_sur        temps_exec
----------  ----------  ----------  ----------       ----------
1           li          rectangles  nyarlathotep165  2638
2           li          rectangles  shubniggurath28  736
3           li          trapezes    nyarlathotep165  4842
...
2154        Ei          puiseux     nyarlathotep145  2766
2155        aleatoire   BBS         azathoth145      524
```

❏ **Q25** – Expliquer pourquoi il n'est pas possible d'utiliser l'attribut `nom` comme clé primaire de la table `fonctions`.

❏ **Q26** – Écrire des requêtes `SQL` permettant de :

1. Connaître le nombre d'ordinateurs disponibles et leur quantité moyenne de mémoire vive.
2. Extraire les noms des PC sur lesquels l'algorithme `rectangles` n'a pas été testé pour la fonction nommée `li`.
3. Pour la fonction nommée `Ei`, trier les résultats des tests du plus lent au plus rapide. Pour chaque test retenir le nom de l'algorithme utilisé, le nom du pc sur lequel il a été effectué et la puissance du PC.

Fin de l'épreuve.

Correction détaillée

✰ Corrigé-Q1

```python
from math import sqrt, log, floor, ceil
print(log(0.5))

# ou encore
import math as m
print(m.log(0.5))
```

✰ Corrigé-Q2

```python
def sont_proches(x,y):
    # c'est le test effectué par numpy dans allclose
    atol = 1e-5
    rtol = 1e-8
    return abs(x - y) <= atol + abs(y) * rtol
```

✰ Corrigé-Q3 `mystere(1001, 10)` renvoie 3.

✰ Corrigé-Q4 C'est la partie entière de $\log_{\mathrm{b}}(x)$ qui est aussi le nombre de chiffres moins un de la représentation en base b de x.

✰ Corrigé-Q5 On ne peut pas représenter 10^{-5} de manière exacte car ce n'est pas une puissance de 2. L'accumulation des erreurs lors des additions successives produit l'effet sur `x2`.

✰ Corrigé-Q6 $\simeq \frac{4*10^9}{4} = 10^9$

✰ Corrigé-Q7 On multiplie par 32 la valeur maximale de `N`.

✰ Corrigé-Q8

```python
def erato_iter(N):
    L = [True] * N
    L[0] = False
    for i in range(2, round(sqrt(N))+1):
        if not L[i-1]:
            continue
        for k in range(2 * i, N+1, i):
            L[k-1] = False
    return L
```

✰ Corrigé-Q9 Changer la valeur d'un élément dans une liste Python est en O(1).
Pour chaque nombre premier p trouvé on devra affecter $\frac{N-p}{p}$ éléments dans la liste soit $O(1) * \frac{N-p}{p}$ opérations donc $\simeq O(N) * \frac{1}{p}$ opérations.

- Pour $p = 2$ on fait donc $O(N) * \frac{1}{2}$ opérations.
- Pour $p = 3$ on fait donc $O(N) * \frac{1}{3}$ opérations.
- ...
- Pour $p = D_p$, avec D_p le dernier premier inférieur à $\sqrt{N}$ on fait donc $O(N) * \frac{1}{D_p}$ opérations.

Donc le nombre total d'opérations est :

$$O(N) * \sum_{p<\sqrt{N}, p \text{ premier}} \frac{1}{p}$$

Soit, en admettant le résultat de l'énoncé, une complexité en :

$$O(N) * \ln(\ln(\sqrt{N})) = O(N \ln(\ln(N)))$$

✰ **Corrigé-Q10** Dans une base b arbitraire $n = \lfloor \log_{\mathrm{b}}(N) \rfloor + 1$.
D'où une complexité en $O(\exp(n) * \ln(n))$.

✰ **Corrigé-Q11** $2^N - 1$

✰ **Corrigé-Q12**

```python
import time
def bbs(N):
    p1 = 24375763
    p2 = 28972763
    M = p1 * p2
    temps = time.time()
    usecs = int((temps - floor(temps)) * 1e7)
    xi = usecs # le premier xi reçoit la valeur de la graine
    A = 0
    for i in range(N):
        if xi%2:
            A = A + 2**i
        xi = (xi ** 2) % M
    return(A)
```

✰ **Corrigé-Q13**

```python
def premier_rapide(n_max):
    # La difficulté est de combiner l'intervalle [0, 2**N - 1 ]de bbs avec le
    # n_max d'ici.
    N = ceil(log(n_max, 2))
    p = bbs(N)
    # on peut ne pas sanctionner l'oubli du cas p == 0...
    while p == 0 or p >= n_max or not (
            2 ** (p - 1) % p == 1 and
            3 ** (p - 1) % p == 1 and
            5 ** (p - 1) % p == 1 and
            7 ** (p - 1) % p == 1
            ):
        p = bbs(N)
    return p
```

✰ **Corrigé-Q14**

```python
def stats_bbs_fermat(N, nb):
    faux_premiers = []
    vrai_premier = erato_iter(N)
    for i in range(nb):
        val = premier_rapide(N)
        if not vrai_premier[val - 1]:
            faux_premiers.append(val)
    return len(faux_premiers)/nb, faux_premiers
```

☆ **Corrigé-Q15**

```python
def Pi(N):
    premiers = erato_iter(N)
    Pi_x = 0
    ret = []
    for idx in range(N):
        if premiers[idx]:
            Pi_x += 1
        ret.append([idx + 1, Pi_x])
    return ret
```

☆ **Corrigé-Q16**

```python
def verif_Pi(N):
    valeurs_test = Pi(N)[5392:]
    for x, pi_x in valeurs_test:
        if pi_x <= x / (log(x) - 1):
            return False
    return True
```

Le résultat cité provient de [*].

☆ **Corrigé-Q17** $O(n)$, n étant le nombre de points, soit $\frac{x}{pas} - 1$. Le calcul de n n'est pas demandé.

☆ **Corrigé-Q18** La même complexité pour tous ces algorithmes, éventuellement l'étudiant peut préciser que la constante multiplicative n'est pas la même.

☆ **Corrigé-Q19**

```python
def inv_ln_rect_d(a, b, pas):
    S = 0
    # Toute boucle du type xi = xi + pas devrait être refusée
    # au vu de l'énoncé de la q5.
    max_n = round((b - a) / pas)
    for i in range(max_n):
        xi = a + (i + 1) * pas
        S = S + 1/log(xi)
    return S*pas
```

☆ **Corrigé-Q20**

```python
def li_d(x, pas):
    if x == 1:
        return -float("inf")
    bmin = min(1 - pas, x)
    S = inv_ln_rect_d(0, bmin, pas)
    if x > 1:
            S += inv_ln_rect_d(1 + pas, x, pas)
    return S
```

☆ **Corrigé-Q21** Pour $x \simeq 1.4$ on a $\mathrm{li}(x) \simeq 0$ donc il est normal que l'écart relatif explose étant donné que l'écart absolu est non nul.

[*]. Dusart, Pierre. « Sharper bounds for Ψ, Θ, Π, pk. » Rapport de recherche (1998). Laboratoire d'Arithmétique, de Calcul formel et d'Optimisation ESA - CNRS 6090 `http://www.unilim.fr/laco/rapports/1998/R1998_06.pdf`

☆ **Corrigé-Q22** On se place toujours dans l'hypothèse simplificatrice où $\varepsilon = pas$ et les approximations sont faites au premier ordre.
Sur un intervalle $[1-\varepsilon, 1+\varepsilon]$ avec $\varepsilon \ll 1$, et en admettant que $\frac{1}{ln(1+\varepsilon)} \simeq -\frac{1}{ln(1-\varepsilon)}$ quand $\varepsilon \to 0$, l'intégrale impropre de $\frac{1}{\ln(x)}$ devrait être nulle en première approximation.
Or l'algorithme des rectangles à droite utilise $\frac{1}{\ln(1-\varepsilon)}$ comme dernier point avant $x = 1$ et $\frac{1}{\ln(1+2\varepsilon)}$ comme premier point après $x = 1$. Donc les termes de part et d'autre de $x = 1$ à son voisinage ne s'annulent pas comme escompté car ils ne sont pas répartis symétriquement, d'où l'augmentation de l'écart absolu.
Une explication plus rigoureuse, mais non demandée est donnée ci-dessous.
À l'ordre 1 on a $\frac{1}{\ln(1+\varepsilon)} \simeq -\frac{1}{\ln(1-\varepsilon)}$ quand $\varepsilon \to 0$. Donc quand on intègre $\frac{1}{\ln(x)}$ de part et d'autre du voisinage de $x = 1$ on peut avoir une intégrale proche de 0.
Avec la méthode des rectangles à droite la dernière valeur évaluée avant 1 est $\frac{1}{\ln(1-\varepsilon)}$, la première après 1 est $\frac{1}{\ln(1+2\varepsilon)}$. Le calcul effectué par l'algorithme sur le dernier rectangle avant 1 et le premier après 1 est donc :

$$Intégrale = \varepsilon \frac{1}{\ln(1-\varepsilon)} + \varepsilon \frac{1}{\ln(1+2\varepsilon)}$$

Ou encore en développant à l'ordre 1 :

$$Intégrale \simeq \varepsilon \frac{-1}{\varepsilon} + \varepsilon \frac{1}{2\varepsilon}$$

D'où :

$$Intégrale \simeq -\frac{1}{2}$$

On peut répéter le raisonnement pour l'avant dernier rectangle de part et d'autre de 1 et avoir $Intégrale \simeq -\frac{1}{2} - \frac{1}{6}$, etc. On voit donc clairement que la méthode de quadrature choisie va induire un biais systématique quand on s'approche de 1 et que ce biais, dans les hypothèses simplificatrices du sujet, est indépendant du pas.

☆ **Corrigé-Q23** L'important est de se rendre compte que l'erreur sur l'écart absolu est intrinsèque à la méthode de quadrature utilisée et que diminuer le pas ne fait au mieux que diminuer le problème mais ne le fait pas disparaître (en fait l'erreur est indépendante du pas avec les hypothèses du sujet). Les rectangles à droite sont donc totalement inadaptés. On peut par exemple utiliser les trapèzes, les rectangles centrés, voire combiner les rectangles à droite avant $x = 1$ avec les rectangles à gauche après. L'important est d'utiliser une méthode qui évalue des points symétriquement de part et d'autre de $x = 1$.

☆ **Corrigé-Q24**

```python
def li_dev(x):
    # attention on travaille sur z=log(x) et la série n'est
    # valable que pour z > 0
    if x <= 1  or log(x) > 40:
        return False
    z = log(x)
    MAXIT = 100
    tmp = 1
    ei = 0.577215664901 + log(z)
    for k in range(1, MAXIT + 1): # plus lisible comme ça
        # on n'attend clairement pas un code aussi compact
        # de la part des élèves (il vient des Numerical
        # Recipes in C) mais au minimum il faut qu'ils aient
```

```
        # une variable k_fact qu'ils multiplient par k à chaque
        # itération.
        # Le gros piège est de faire une boucle for qui calcule
        # k! à chaque itération de k. Pour z**k le calcul de
        # puissance est en O(1) sur les réels donc ça va.
        tmp *= z / k
        old = ei
        ei += tmp / k
        if sont_proches(ei, old):
            return ei
    return False
```

☆ **Corrigé-Q25** Une clé primaire doit être unique, or on peut vouloir tester plusieurs fois la même fonction donc on aura des doublons sur cet attribut.

☆ **Corrigé-Q26**

```sql
SELECT COUNT(*), AVG(ram) FROM ordinateurs;

SELECT nom FROM ordinateurs WHERE nom NOT IN (
    SELECT teste_sur FROM fonctions WHERE
    nom="li" AND algorithme="rectangles"
);

SELECT algorithme, teste_sur, gflops FROM fonctions
    JOIN ordinateurs ON fonctions.teste_sur=ordinateurs.nom
    WHERE fonctions.nom="Ei" ORDER BY temps_exec DESC;
```

Note : dans la deuxième requête SQL, on peut aussi utiliser "EXCEPT".