Correction concours blanc informatique 2021

Codage autour des nombres premiers

Partie I. Préliminaires

Q01 Voici par exemple l'affectation aux variables a, b, c, d de la valeur de chacune des fonctions demandées en $0,5$ si l'on crée un alias au module *math*

```
import math as m

a = m.log(0.5)
b = m.sqrt(0.5)
c = m.floor(0.5)
d = m.ceil(0.5)
```

ou bien, si l'on importe toutes les fonctions de ce module

```
from math import *

a = log(0.5)
b = sqrt(0.5)
c = floor(0.5)
d = ceil(0.5)
```

Q02 Rappelons que la condition « $x \leqslant y$ » est un booléen qui vaut *True* si $x \leqslant y$ et *False* sinon. On peut alors écrire aisément la fonction demandée.

```
def sont\_proches (x,y) :
    atol = 1e-5
    rtol = 1e-8
    return abs(x-y) <= atol + abs(y) * rtol
```

Q03 *mystere(1001,10)* nécessite le calcul de *1001/10* (car $1001 \geqslant 10$) qui vaut 100 et calcul *1+mystere(100,10)*.
mystere(100,10) nécessite le calcul de *10/10* (car $100 \geqslant 10$) qui vaut 10 et calcul *1+mystere(10,10)*.
mystere(10,10) nécessite le calcul de *1001/10* (car $10 \geqslant 10$) qui vaut 1 et calcul *1+mystere(1,10)*.
mystere(1,10) renvoie 0 car $1 < 10$.
Par conséquent, *mystere(1001,10)* renvoie $1 + (1 + (1 + 0)) = 3$.
Si $b = 1$ alors, si $x \geqslant b = 1$, *mystere(x,b)* appele *mystere(x,b)* (car $x/b = x$) donc la fonction *mystere(x,b)* ne s'arrête jamais (boucle infinie).

Q04 On suppose désormais que $b \geqslant 2$. Comme la suite $(b^n)_{n\in\mathbb{N}}$ est strictement croissante et tend vers $+\infty$, les intervalles $[[1, b[[$, $\left[\left[b, b^2\right[\right[$, $\left[\left[b^2, b^3\right[\right[$, etc, c'est-ÃÂ -dire les intervalles $\left(\left[\left[b^n, b^{n+1} - 1\right[\right[\right)_{n\in\mathbb{N}}$ forment une partition de $\mathbb{N}^*$.
Si $x \in [[1, b[[$ alors *mystere(x,b)* renvoie 0 (car $x < b$).
Si $x \in \left[\left[b, b^2\right[\right[$ alors

mystere(x,b) = 1 + mystere(x/b,b) = 1 + 0 =1

(car $x/b \in [[1, b[[$).
Si $x \in \left[\left[b^2, b^3\right[\right[$ alors

mystere(x,b) = 1 + mystere(x/b,b) = 1 + 1 =2

(car $x/b \in \left[\left[b, b^2\right\right[$).
Plus généralement, si $x \geqslant 1$, il existe un unique entier N tel que $x \in \left[\left[b^N, b^{N+1}\right\right[$ alors *mystere(x,b)* vaut N (preuve par récurrence si cela vous intéresse mais le sujet ne le demande pas). Il reste à expliciter N en fonction de x. Pour cela, on utilise l'encadrement liant x, b et N.

$$\begin{aligned} b^N &\leqslant x < b^{N+1} \underset{\ln\nearrow}{\Leftrightarrow} N\ln(b) \leqslant \ln(x) < (N+1)\ln(b) \\ \underset{\div \ln(b)>0}{\Leftrightarrow} N &\leqslant \frac{\ln(x)}{\ln(b)} = \log_b(x) < N+1 \Leftrightarrow N = \lfloor \log_b(x) \rfloor \end{aligned}$$

(par définition de la partie entière). Autrement dit, *mystere(x,b)* renvoie $\lfloor \log_b(x) \rfloor$. C'est aussi le nombre de chiffres de x en base b puisque

- si $x \in [\![1, b[\![$, x a un chiffre en base b ;
- si $x \in \left[\left[b, b^2\right\right[$, x s'écrit $x = \overline{x_1 x_2}$ avec $x_1 \neq 0$ donc il a deux chiffres en base b ; etc.

Remarque : Cette question est très discriminante car elle nécessite initiative (tester pour diverses valeurs de x), intuition (faire le lien avec le nombre de chiffres en base b) et calcul mathématique (faire le lien avec le logarithme en base b et la partie entière) pour y répondre. Un candidat indiquant que *mystere(x,b)* renvoie le nombre de chiffres de x en base b sera fortement valorisé.

Q05 La valeur renvoie pour *x1* est celui de la dernière valeur de la boucle :

$$(99\,999 + 1) * 10^{-5} = 10^5 * 10^{-5} = 1$$

car le produit des flottants dont les mantisses sont égales est simplement le produit des exposants d'où l'exactitude du produit.
Par contre, pour *x2*, il s'agit d'additions répétées. A partir d'un certain rang, x et *pas* n'ont plus le même exposant. Dans ce cas, Python choisit un exposant commun e_{commun} et modifie les mantisses (i.e. ils les écrits sous la forme $\varepsilon * m * 10^{e_{\text{cimmun}}}$ avec $\varepsilon = \pm$ mais m n'est plus nécessairement un nombre entre 1 et 10, non inclus donc l'un des deux mantisses a ses décimales qui se décalent donc certaines vont être définitivement perdues), il ajoute ensuite les mantisses puis renormalise la somme de ces deux là (i.e. l'écrit sous la forme $\varepsilon * m * 10^e$, où ε est le signe, m un flottant entre 1 et 10, exclus, e l'exposant entier, c'est-à-dire qu'il redécale probabablement la somme des mantisses d'où de nouvelles erreurs), ce qui crée des erreurs d'arrondis, puis renvoie $\varepsilon * m * 10^{e+e_{\text{commun}}}$. Ainsi, la valeur renvoyée pour *x2* n'est pas exactement 1 (l'erreur en 10^{-11}).

Remarque : Cette question teste les connaissances du candidat concernant les flottants à virgule flottante (qui est une thématique récurrente). Il n'est pas attendu une explication théorique mais plutôt l'idée de la notion d'erreur d'arrondi. Cette question est discriminante.

Partie II. Génération de nombres premiers

II.a Approche systématique

Q06 Un booléen est codé sur 32 bits donc sur $\frac{32}{8} = 4$ octets. Ainsi, avec 4 Go c'est-à-dire $4 * 10^9$ octets, on peut stocker 10^9 booléens donc N vaut au plus 1 milliard (environ car on néglige les marqueurs nécessaires pour indiquer qu'il s'agit d'une liste et son mode de navigation).

Remarque : Question élémentaire mais qui s'avère en pratique discriminante car un nombre important de candidats ne savent pas faire de type de calculs élémentaires (sans compter les liens entre bit, octet, kilo ou méga ou giga-octets). N'hésitez pas à travailler de type de petites questions assez rituelles.

Q07 En codant un booléen sur 1 bit (par exemple, 0 pour *False* et 1 pour *True*), on divise par 32 le coût de stockage donc on multiple par 32 le nombre N.

Remarque : Même type de remarque qu'à la question précédente.

Q08 On construit une liste L à $N+1$ éléments dont chacun vaut *True*. Pour tout entier i entre 0 et N, $L[i]$ encodera si i est un nombre premier ou non donc $i = 0$ et $i = 1$ ne seront pas exploités (d'où l'apparation des deux numéros supplémentaires, cela évitera les décalages pénibles pour passer de l'entier 1 à $L[0]$ et réciproquement, etc).
Si $L[i]$ n'est pas marqué faux, c'est que $L[i]$ vaut True alors, les multiples de i autres que i sont de la forme $i * k$ avec

$$\left\{ \begin{array}{c} 2 \leqslant k \text{ et} \\ i * k \leqslant N \end{array} \right. \Leftrightarrow \left\{ \begin{array}{c} 2 \leqslant k \text{ et} \\ k \leqslant \frac{N}{i} \end{array} \right. \Leftrightarrow \left\{ \begin{array}{c} 2 \leqslant k \text{ et} \\ k \leqslant \left\lfloor \frac{N}{i} \right\rfloor \end{array} \right.$$

(car k est un entier). On peut maintenant implémenter littéralement l'algorithme proposé du sujet.

```
def erato_iter(N):
    L = [True for i in range(N+1)]
    L[0], L[1] = False, False
    for i in range(2,floor(sqrt(N))+1):
        if L[i]:
            for k in range(2,floor(N/i)+1):
                L[i*k] = False
    return L
```

Voici une autre version si on ne sait pas déterminer la dernière valeur de i dans la boucle *for*.

```
def erato_iter(N):
    L = [True for i in range(N+1)]
    L[0], L[1] = False, False
    for i in range(2,floor(sqrt(N))+1):
        if L[i]:
            k = 2
            while i*k < N+1:
                L[i*k] = False
                k = k + 1
    return L
```

Q09 $\lfloor\sqrt{N}\rfloor \sim \sqrt{N}$ quand N tend vers $+\infty$ car :

$$\lfloor\sqrt{N}\rfloor \leqslant \sqrt{N} < \lfloor\sqrt{N}\rfloor + 1.$$

Et :

$$1 \leqslant \frac{\sqrt{N}}{\lfloor\sqrt{N}\rfloor} < 1 + \frac{1}{\lfloor\sqrt{N}\rfloor}.$$

Et quand N tend vers $+\infty$, $\dfrac{\sqrt{N}}{\lfloor\sqrt{N}\rfloor}$ tend vers 1.

Évaluons la complexité de *erato_iter(N)*.
La création de la liste L de $N+1$ booléens (L représente *liste_bool* de l'algorithme d'Erathostène) a une complexité en $O(N)$. Le calcul de $\left\lfloor\sqrt{N}\right\rfloor$ est présumé à coût constant, en $O(1)$.

La boucle *for* sur i se répète $\left\lfloor\sqrt{N}\right\rfloor$ fois. À l'intérieur d'une telle itération, le test booléen *L[i]* a un coût constant, en $O(1)$ et, i est un nombre premier (c'est-à -dire si le test est réussi), la boucle *for* sur k s'itère $\left\lfloor\dfrac{N}{i}\right\rfloor - 1$ fois et, chaque itération de cette boucle a un coût constant (une opération arithmétique *i*k* et une affectation)) donc une complexité en

$$O(1) + O\left(\left\lfloor\frac{N}{i}\right\rfloor - 1\right) = O\left(\left\lfloor\frac{N}{i}\right\rfloor\right) = O\left(\frac{N}{i}\right) \leqslant D\frac{N}{i}$$

pour une constante D convenanble et pour N assez grand.
Ainsi, la boucle *for* sur tous les entiers i non premiers (il y a au plus $\left\lfloor\sqrt{N}\right\rfloor$) a une complexité au plus en

$$\left\lfloor\sqrt{N}\right\rfloor O(1) = \sqrt{N}O(1) = O\left(\sqrt{N}\right)$$

(car $\left\lfloor\sqrt{N}\right\rfloor \underset{N\to+\infty}{\sim} \sqrt{N}$).

Par contre, la boucle *for* sur les i premier entre 1 et $\left\lfloor \sqrt{N} \right\rfloor$ a une complexité au plus en

$$\sum_{\substack{i \in [\![1, \lfloor \sqrt{N} \rfloor]\!] \\ i \text{ premier}}} D\frac{N}{i} = DN \sum_{\substack{i \in [\![1, \lfloor \sqrt{N} \rfloor]\!] \\ i \text{ premier}}} \frac{1}{i}$$

$$\underset{N \to +\infty}{\sim} DN \ln\left(\ln\left(\sqrt{N}\right)\right) \text{ (d'après la formule (1))}$$

$$= DN \ln\left(\frac{1}{2}\ln(N)\right) = DN\left(\ln\left(\frac{1}{2}\right) + \ln(\ln(N))\right) \underset{N \to +\infty}{\sim} DN \ln(N) = O(N \ln(\ln(N))).$$

Au final, la complexité du crible est en

$$O(N) + O\left(\sqrt{N}\right) + O(N \ln(\ln(N))) = O(N \ln(\ln(N))).$$

Remarques : Un candidat faisant varier i entre 2 et N dans son code informatique obtiendra aussi une complexité en $O(N \ln(\ln(N)))$ et ne sera pas sanctionné car son calcul est cohérent avec son code (si celui remplit à peu près sa mission).
Un candidat indiquant que chaque itération de la boucle *for* (sur i) a une complexité en $O(N)$ (ce qui n'est pas mais est une majoration un peu brutale) obtiendra une complexité totale en $O\left(N^2\right)$. Il n'obtiendra pas l'intégralité des points de cette question mais raisonnablement au moins la moitié de ceux-ci.
Par contre, il est attendu des candidats qu'il explique, avec quelques phrases bien choisies, le calcul de complexité en ne se limitant pas au nombre d'itérations des boucles (mais aussi la complexité, par exemple, d'une itération donnée).

Q10 Choisissons la base $b = 2$ (mais le raisonnement ne change pas pour une base quelconque). Si N possède p chiffres dans son écriture en base $b = 2$, c'est qu'il est compris entre

$$\begin{aligned} \overline{10...0}^b &= 1b^{p-1} + 0b^{p-2} + \cdots + 0b^0 = b^{p-1} \text{ et} \\ \overline{(b-1)(b-1)...(b-1)}^b &= \sum_{k=0}^{p-1} (b-1) b^k = \sum_{k=0}^{p-1} \left(b^{k+1} - b^k\right) = b^p - b^0 \end{aligned}$$

donc

$$b^{p-1} \leqslant N < b^p \underset{\ln \nearrow}{\Rightarrow} (p-1)\ln(b) \leqslant \ln(N) \leqslant p\ln(b)$$

On en déduit la majoration suivante :

$$N \ln(N) < b^p p \ln(b) = O(pb^p) \Rightarrow O(N \ln(N)) = O(pb^p).$$

Autrement dit, la complexité du critère d'Erathostène est sur-exponentiel en fonction du nombre p de ses chiffres en base b (c'est-à-dire que sa complexité est infiniment grande devant celle de l'exponentielle).

Q11 Par construction, la valeur finale de A après $N-1$ itérations est :

$$A = 0 + \sum_{\substack{i \in \{1,..,N-1\} \\ \text{avec } x_i \text{ impair}}} 2^i.$$

Ainsi, si tous les x_i sont supposés impairs, la valeur de A est :

$$A = A_{\text{impair}} = \sum_{i=1}^{N-1} 2^i = 2^1 \times \frac{2^{N-1} - 1}{2 - 1} = 2^N - 2.$$

qui n'est pas un nombre premier ! Par contre, tout nombre A généré par l'algorithme Blum Blum Shub avec $N-1$ intérations est inférieur à A_{impair} donc A est strictement inférieur à 2^N.

Q12 Il est nécessaire d'importer la bibliothèque *random* pour pourvoir utiliser la fonction *randint* afin de créer la graine (ligne 1).
Cette graine x_0 (noté *xi* dans le corps du code suivant) ne peut être nul ni être égal à M (sinon $x_1 = x_0^2 mod(M)$ vaut 0 et la suite $(x_n)_n$ sera identiquement nulle à partir du rang 1) ni égal à 1 (sinon la suite est constante et peu d'aléa va en sortir !). Plus généralement, cette graine doit être un nombre premier avec M mais le sujet ne le mentionne pas.
À la ligne 8, on crée un nombre entier entre 1 et $M-1$.
À la ligne 11, le caractère impair s'obtient par le reste de la division euclidienne de *xi* par 2. A la ligne 14, on utilise le code habituel pour les suites $u_{n+1} = f(u_n)$.

```
import random as rd


def bbs(N):
    p1 = 24374763
    p2 = 28972763
    M = p1*p2
    xi = rd.randint(2,M-1)
    A = 0
    for i in range(N):
        if xi%2 == 1:
            A = A + 2**i
        xi = ((xi)**2)%M
    return A
```

Remarques : Les correcteurs ne sanctionneront pas les candidats sur le choix de la graine (car le sujet ne le précise pas) c'est-à-dire qu'il soit nul ou d'autres critères.
Question sans difficulté particulière.

Q13 Si l'on note $d = \lfloor \log_2(x) \rfloor$ est le chiffre de $n_{\max}$ en base 2 alors $2^{d-1} \leqslant n_{\max} < 2^d$ (cf. la question 10) et A est le nombre construit par l'algorithme BBS après $d-1$ itérations, d'après la question 11, on a

$$A < 2^{d-1} \leqslant n_{\max}$$

donc A est un nombre aléatoire strictement inférieur à $n_{\max}$. On teste si A est différent de 0 ou 1 (sinon il n'est pas trop premier) et, dans ce cas, on met en place le test de primalité de Fermat pour $a \in \{2, 3, 5, 7\}$ (comme souhaité par le sujet). Si cela n'est pas le cas, on recréé un tel nombre A avec $d-1$ itérations et on le reteste. On poursuit ainsi jusqu'à ce que A soit (probablement) un nombre premier.
Pour commencer, on crée une fonction *test_fermat(L,p)* qui teste si p vérifie le critère de Fermat pour tout les éléments a de L (c'est-à -dire si $a^{p-1} = 1\ [p]$). On élimine d'emblée le cas où $p = 0$ ou $p = 1$ (valeurs que peut prendre A).

```
def test_fermat(L,p):
    if p in [0,1]:
        return False
    for a in L:
        if (a**(p-1))%p != 1:
            return False
        return True
```

On peut alors écrire aisément l'algorithme.

```
def premier_rapide(n_max):
    d = floor(log(n_max,2))
 $\# $ \textit{on peut taper} d = mystere(n\_max,2)
    base = [2,3,5,7]
    A = bbs(d-1)
    while not test_fermat(base,A):
        A = bbs(d-1)
    return A
```

Q14 Si l'on ne connaît pas *erato_iter*, on commence par écrire une fonction *est_premier(n)* qui renvoie *True* si n est un nombre premier et *False* sinon. Pour cela, on remarque que si d est un diviseur de n alors $\frac{n}{d}$ est aussi un diviseur de n (car $n = d \times \frac{n}{d}$). Ainsi, si n possède un diviseur d, n possède un diviseur infé rieur à $\sqrt{n}$ (si $d \leqslant \sqrt{n}$ alors d convient, si $d > \sqrt{n}$ alors $\frac{n}{d} < \frac{n}{\sqrt{n}} = \sqrt{n}$ convient). Ainsi, on teste si l'un des entiers d entre 2 et $\sqrt{n}$ divise n. Si c'est le cas, n n'est pas premier, sinon n est nécessairement premier.

```
def est_premier(N):
    if N <= 1:
        return False
    for d in range(2,floor(sqrt(N))+1):
        if N%d == 0:
            return False
        return True
```

On peut alors écrire la fonction souhaitée. On génère *nb* nombres par la fonction *premier_rapide(N)*, on teste chacun de ces nombres par la fonction *est_premier*, on compte le nombre *nbfaux* ceux qui ne sont pas premiers et on renvoie la proportion $\frac{nbfaux}{nb}$ qui est le taux d'erreur.

```
def stats_bbs_fermat(N,nb):
    nbfaux = 0
    for i in range(nb):
        if not est_premier(premier_rapide(N)):
            nbfaux = nbfaux + 1
    return nbfaux/nb
```

Version en utilisant *erato_iter* :

```
def stats_bbs_fermat(N,nb):
    nbfaux = 0
  premiers = erato\_iter(N)
  faux = [ ]
  for i in range(nb):
      p = premier\_rapide(N)
          if not premiers[p-1] :
              nbfaux = nbfaux + 1
          return nbfaux/nb, faux
```

Partie III. Compter les nombres premiers

Q15 On utilise la liste *liste_bool* associée aux entiers entre 1 et n. Chaque fois que c'est le cas en une position i donnée, pour chaque entier $j \geqslant i$, on augmente d'une unité le compteur comptant les nombres premiers inférieurs à j (par exemple, si 3 est premier, il est comptabilisé dans les nombres premiers inférieurs à 3 mais aussi à 4, ÃÂ 5, .. , à n).

Pour l'implémentation, voici comment on procède. On appelle *eroto_iter(n)* et on stocke sa valeur dans un tableau *table*.
On construit également un tableau *nb* tel que, pour tout entier $i \geqslant 1$, la liste $[i, \pi(i)]$ est stockée dans la cellule d'indice $i-1$ (donc dans *nb[i-1]*). Par exemple, *table[3]* contiendra la liste $[4, \pi(4)]$.
Voici le code correspondant.

```
def Pi(N):
    table = eroto_iter(N)
    nb = [[k,0] for k in range(len(table))]
    for i in range(1,len(table)):
        if table[i]:
            for j in range(i,len(table)):
                nb[j][1] = nb[j][1] + 1
    return nb
```

Q16 On stocke la liste produit par *Pi(N)* dans la table *test*. Rappelons que, pour tout entier $n \geqslant 1$, $\pi(n)$ est stocké à l'indice $n-1$ de cette table (qui contient en cet indice la liste $[n, \pi(n)]$. On parcourt tous les entiers entre 5393 et N. Si $c_{n-1} = \pi(n)$ est inférieur ou égal à $\dfrac{n}{\ln(n)-1}$, on renvoie *False* (la propriété n'est pas vérifiée pour cet entier). Si la boucle s'achève, tous les tests sont réussis et on renvoie *True*. Voici le code (sous hypothèse que $N \geqslant 5393$, sinon il faut rajouter au début un test pour vérifier que $N \geqslant 5393$ et renvoyer directement *None* si cela n'est pas le cas).

```
def verif_Pi(N):
test = Pi(N)
for n in range(5393,N):
        if test[n-1][1] <= n/(log(n)-1):
                return False
        return True
```

Remarque : Le sujet ne dit pas que $\dfrac{n}{\ln(n)-1} < \pi(n)$ est faux si $n < 5393$ car ... ce n'est pas vrai. Par exemple, pour $n = 2$, $\pi(2) = 1$ et $\dfrac{2}{\ln(2)-1}$ est manifestement négatif donc l'inégalité est vérifiée.

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

Estimation de li par quadrature numérique

Q17 Rappelons l'approximation de $\displaystyle\int_a^b f(t)\,dt$ par la méthode des rectangles à droite. Notons h le pas (la largeur des rectangles). On découpe l'intervalle $[a,b]$ en une union d'intervalles $([a_k, a_{k+1}])_{0\leqslant k\leqslant N}$ avec $a_0 = a$, $a_N = b$, et $a_{k+1} = a_k + h$ pour tout $k < N-1$ et l'écart entre a_{N-1} et $a_N = b$ est inférieur ou égal à 1 (on ne peut pas toujours subdiviser exactement l'intervalle donnné avec tout pas). L'approximation recherchée est

$$I_h = \sum_{k=1}^{N} hf(a_k)$$

(somme des rectangles à droites.)
On peut alors évaluer la complexité de l'approximation. Elle nécessite

- la construction des $N+1$ points $(a_k)_{0\leqslant k\leqslant N}$ (de complexité $O(N)$ car ils sont construits via une boucle *for* ayant environ itérations $a_{k+1} = a_k + h$ et des tests booléens $a_k < b$) ;
- $N+1$ évaluations de f en ces points $(a_k)_{0\leqslant k\leqslant N}$ (de complexité $O(N)$) ;
- de la multiplication et de la sommation de $N+1$ rÃÂ©els (les $hf(a_k)$) (de complexité $O(N)$.

Si l'on considère que le pas h est la variable d'entrée, il faut la lier avec l'entier N. Voici comment procéder. Comme la réunion des $N-1$ premiers intervalles $[a_0,a_1],..,[a_{N-2},a_{N-1}]$ est $[a,a_{N-1}]$ qui est inclus dans l'intervalle $[a,b]$, la somme de leurs longueurs est inférieur à celui de $[a,b]$. Comme ils sont tous de longueur h, on a l'inégalité $(N-1)h \leqslant b-a$. En outre, le dernier intervalle est de longueur strictement inférieur àÂ h, l'intervalle $[a,b]$ a une longueur strictement inférieur à Nh d'où l'encadrement suivant :

$$\begin{aligned}(N-1)h &\leqslant b-a < Nh \Leftrightarrow N-1 \leqslant \frac{b-a}{h} < N\\ &\Rightarrow N-1 = \left\lfloor \frac{b-a}{h} \right\rfloor \Leftrightarrow N = \left\lfloor \frac{b-a}{h} \right\rfloor + 1\end{aligned}$$

(par définition de la partie entière). En particulier, si h tend vers 0, N tend vers $+\infty$ et on a l'équivalent suivant :

$$N \underset{h\to 0}{\sim} \left\lfloor \frac{b-a}{h} \right\rfloor \underset{h\to 0}{\sim} \frac{b-a}{h} = O\left(\frac{1}{h}\right).$$

Au final, la complexité totale est en

$$O(N) + O(N) + O(N) = O(N) = O\left(\frac{1}{h}\right).$$

Commentaires : La méthode des rectangles bien connu des candidats et la complexité est évidemment en $O(N)$. Une telle réponse sera valorisée si le candidat énonce clairement la variable d'entrée (ici f, a, b et N). Les candidats parvenant à faire le lien avec le pas h, ou du moins, en indiquant que N est de l'ordre de grandeur en $\frac{1}{h}$ obtiendra assurément la totalité des points à cette question.
Il s'agit d'une question de cours qui est bien réussie en général. Remarquons que cette question intervient relativement régulièrement à ce concours.

Q18 Avec les notations de la réponse à la question précédente, l'approximation de $\int_a^b f$ par la méthode

- des rectangles centrés est $\sum_{k=0}^{N-1} hf\left(\frac{a_k + a_{k+1}}{2}\right)$ (le rectangle ayant pour hauteur le milieu de chaque segment $[a_k, a_{k+1}]$)
- des trapèzes est $\sum_{k=0}^{N-1} h\left(\frac{f(a_k) + f(a_{k+1})}{2}\right)$ (moyenne des aires des rectangles gauche et droit).

Manifestement, la complexité est la même que précédemment c'est-àÂ -dire en $O(N)$ si la variable est N et en $O\left(\frac{1}{h}\right)$ si la variable est h.

Commentaires : Mêmes remarques qu'àÂ la question précédente.

Q19 On conserve les notations de la réponse à la question 17. On crée au fil de l'eau les différentes valeurs des a_k via une variable notée x dans le code suivant. Le compteur stockant la somme sera noté S**.** On ajoute à x ajoute le pas *pas* tant que $x < b$ et on ajoute alors $\frac{1}{\ln(x)}$ àÂ S. Ãâ¬$la fin del'itération, onregardesion$

```
def inv_ln_rect_d(a,b,pas):
    x = a
    S = 0
    while x < b:
        x = x + pas
        S = S + 1/(log(x))
    if x > b:
        S = S + 1/(log(b))
    return pas*S
```

On peut aussi utiliser *Numpy* et la fonction *np.arange(a,b,pas)* qui crée la liste des $(a_k)_k$ (i.e. la boucle *while* du code précédent). Il faut penser à ne pas prendre le premier point a_0. Voici le code associé.

```
def inv_ln_rect_d(a,b,pas):
    X = np.arange(a,b,pas)
    S = 0
    for i in range(1,len(X)):
        S = S + 1/log(X[i])
    return pas*S
```

Commentaires : Il s'agit d'une question de cours qui ne doit pas poser de problème particulier aux candidats. Les deux impléentations de ce corrigé seront acceptés au concours en préférence (de la part des correcteurs) pour l'une ou l'autre. Le sujet indique que $b - a$ est un multiple de *pas* donc le test « *if* $x > b$ » est inutile (mais je l'ai mis pour le plus général possible).

Q20

D'après l'énoncé, (III.b, de la formule (4) jusqu'avant la question 17) l'approximation de $Li(x) = \int_0^x \frac{dt}{\ln(t)}$ sera $\int_{pas}^x \frac{dt}{\ln(t)}$ si $x < 1$ et $\int_0^{1-pas} \frac{dt}{\ln(t)} + \int_{1+pas}^x \frac{dt}{\ln(t)}$ si $x > 1$. Chacune de ces intégrales étant des approximations par la méthode des rectangles à droite. Pour finir, si $x = 1$, $Li(x)$ sera approximer par $\int_0^{1-pas} \frac{dt}{\ln(t)}$. Voici le code répondant à la question.

```
def li_d(x,pas):
    if x < 1 :
        return inv_ln_rect_d(0,x,pas)
    elif x > 1 :
        return inv_ln_rect_d(0,1-pas,pas)+inv_ln_rect_d(1+pas,x,pas)
    else:
        return 0
```

Commentaires : La seule difficulté de cette question est d'opérer la disjonction de cas (via *if then else*). Cette question s'avère discriminante pour cette raison.

Q21 D'après l'énoncé, comme $Li(x) = Ei(\ln(x))$. On pose $t = \ln(x)$ alors il faut calculer $Ei(t)$. D'apr ès l'énoncé, on fait l'approximation de $Ei(x)$ par $Ei_n(t)$ avec n de sorte qu'il s'agit du premier entier $k \geqslant 1$ tel que $Ei_k(t)$ et $Ei_{k-1}(t)$ sont proches au sens de la fonction *sont_proches* de la question 2 (i.e. *est_proche*$(Ei_k(t), Ei_{k-1}(t))$ renvoie *True*).

$$est\_proche(Ei_k(t), Ei_{k-1}(t) = \left|\frac{t^k}{k \times k!}\right|.$$

En outre, on choisit cet entier k parmi les entiers inférieurs à $MAXIT = 100$ (d'après l'énoncé). L'entier n étant fixé, il faut calculer la somme

$$\sum_{k=1}^{n} \frac{x^k}{k \times k!}.$$

Voici une version naive et réguliérement rencontrée : calcul de $n!$ (par itération ou récursive), calcul de x^k grace àÂ la fonction *x**k*.

```
def fact(n):
    if n <= 1:
        return 1
    else:
        return n*fact(n-1)

def li_dev(x):
    MAXIT = 100
    t = log(x)
    gamma = 0.577215664901
    ancien = gamma + log(abs(t)) + t
    for k in range(2,MAXIT+1):
        nouveau = ancien + t**k/(k*fact(k))
    if sont_proches(ancien,nouveau):
        return nouveau
    else:
        ancien = nouveau
        return nouveau
```

Malheureusement, la complexite est en $O(MAXIT^2)$. En effet, la boucle *for* (par rapport à k) s'exécute $MAXIT$ fois au plus. A chacune de ces itérations, le calcul de *x**k* a une complexité en $O(k)$ (il faut calculer les k multiplications de x avec lui-même, il s'agit d'une complexité dite cachée

mais il est attendu des candidats de connaitre de tels coûts dans les cas usuels), le calcul de *fact(k)* a aussi une complexité en $O(k)$, les tests booléens et autres opérations arithmétiques ont une complexité en $O(1)$. Ainsi, chaque itération a une complexité en

$$O(k) + O(k) + O(1) = O(k) = O(MAXIT)$$

(car $k \leqslant MAXIT$) donc la complexité totale est bien en est est en

$$MAXITO(MAXIT) = O\left(MAXIT^2\right).$$

Pour arriver à une complexité linéaire en *MAXIT*, on peut songer à l'algorithme d'Horner mais cela n'est pas possible ici car l'entier n n'est pas prédéterminé. Pour contourner le problème, on crée trois variables. L'une *puiss* stockant les puissances de t (t^k), l'autre *fact* stockant les factorielles ($k!$), *ancien* stockant $Ei_{k-1}(t)$ et *nouveau* stockant $Ei_k(t)$. Remarquons les relations suivantes :

$$\begin{aligned} t^k &= t \times t^{k-1}, k! = k \times (k-1)!, Ei_k(t) = Ei_{k-1}(t) + \frac{t^k}{k \times k!} \\ t^1 &= 1, 1! = 1, Ei_1(t) = \gamma + \ln(t) + t, Ei_2(t) = E_1(t) + \frac{t^2}{4}. \end{aligned}$$

Bien entendu, on utilise une boucle *while* pour déterminer n et même à chaque étape les variables *puiss*, *fac* et *Ei*.

```
def li_dev(x):
    MAXIT = 100
    t = log(x)
    gamma = 0.577215664901
    ancien = gamma + log(abs(t)) + t
    k = 2
    fact = 2
    puiss = t**2
    nouveau = ancien + puiss/(k*fact)
    while (not sont_proches(ancien,nouveau)) and k <= MAXIT:
            k = k+1
            puiss = t*puiss
            fact = k*fact
            ancien, nouveau = nouveau, ancien + puiss/(k*fact)
    return nouveau
```

La boucle *while* s'exécute au plus *MAXIT* fois. À chaque itération de celle-ci, les instructions s'exécute en un nombre borné d'opérations, en $O(1)$ donc la complexité totale est en $O(MAXIT)$.

Commentaires : La construction de la fonction *def li_dev(x)* est assez classique et proche du cours, elle est réussie par une grande partie des candidats, du moins dans sa réponse naive (la première réponse proposée par le corrigé).
Rappelons que l'appel à des bibliothèques pour calculer $k!$ sera sanctionnée et les candidats utilisant la fonction *sum* (qui est licite) devront connaitre sa complexité (en $O(n)$ si elle est appliquée à une liste à n éléments) car son coût est cachée mais c'est un attendu à bac +2.
Néanmoins, la question exige la complexité de la fonction donc un candidat qui produit la première réponse et n'en détermine pas la complexité n'aura aucun point à cette question. Par contre, s'il évalue sa complexité et la justifie (partiellement) en $O\left(MAXIT^2\right)$, il aura une fraction des points (évaluation de la complexité et honneteé scientifique s'il reconnait que la complexité est trop grande) mais il est probable que cela n'exédera pas la moitié de ceux associées àÂ cette question.
Par contre, s'il propose la seconde version de complexité linéaire en *MAXIT*, il n'a pas besoin de la justifier et obtiendra tous les points associés (sous réserve que son code soit correct).
À nouveau, il s'agit d'une question qui sélectionne les candidats ayant un peu de recul sur la programmation informatique et des connaissances significatives en complexité.

Q22 L'attribut *nom* ne peut être clé primaire de la table *fonction* car les 3 premiers enregistrements de l'exemple de table *fonction* possède le même attribut *nom* (ici *rectangles)* alors que les enregistrements sont distincts.

Commentaires : Il s'agit d'une question de cours sans aucune difficulté.

Q23

1. Il suffit d'utiliser les fonctions d'agrégation *COUNT* et *AVG* sur les attributs souhaités.

```
SELECT count(nom),AVG(ram) from ordinateurs
```

2. On commence par récupérer les noms des ordinateurs ayant utilisé l'algorithme *rectangle* et la fonction *li*. Pour cela, on croise les tables *ordinateur* et *fonction* selon la condition d'égalité des attributs *nom* et *teste_sur* et on projette selon l'attribut *nom* (dans la table *ordinateurs*) sous condition que l'algorithme *rectangle* et la fonction *li* aient été choisis.

```
SELECT o.nom
from ordinateurs as o join fonctions as f
ON o.nom = f.teste_sur
wHERE f.algorithme ="rectangles" and f.nom = "li"
```

Il suffit alors de créer la table des noms extrait de la table *fonction* et de lui enlever tous les enregistrements prÃÂ©sents dans la table créé ci-dessus grâce à la commande *EXCEPT*

```
SELECT nom from ordinateurs
EXCEPT
SELECT o.nom
from ordinateurs as o join fonctions as f
ON o.nom = f.teste_sur
wHERE f.algorithme ="rectangles" and f.nom = "li"
```

3. On croise les tables *ordinateurs* et *fonctions* selon la condition d'égalité des attributs *nom* et *teste_sur*, on ne conserve que les enregistrements associé àÂ la fonction *Ei*, on projette selon les attributs *algorithme*, nom du pc et *gflops* puis on trie le tout (grâce àÂ la commande *ORDER BY*) par ordre décroissant du temps d'exécution (grâce à la commande *DESC*).

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

Commentaires : La première requête est simple et bien réussie par la plupart des candidats.
La deuxième requête est sélective car elle est bien plus complexe et requiert soit la connaissance de la commande *EXCEPT*, soit une bonne maitrise des jointures.
La troisième requête est discriminante mais elle est classique et vous l'avez probablement rencontré àÂ plusieurs reprises durant votre scolarité en CPGE (À connaître donc).