Table des matières

1 **Définitions, rappels, notations** **1**

2 **Préliminaires** **2**

3 **Génération de nombres premiers** **3**
3.1 Approche systématique 3

4 **Génération rapide de nombres premiers** **4**

5 **Compter les nombres premiers** **5**
5.1 Calcul de (n) via un crible 5
5.2 Calcul approché via une intégrale généralisée 6
5.2.1 Estimation de *li* par quadrature numérique 6
5.2.2 Analyse des résultats de `li_d` 6
5.3 Estimation de *li* via *Ei* 7

6 **Évaluation des performances (BDD)** **7**

Autour des nombres premiers

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.

1 Définitions, rappels, notations

- On appellera « nombre » toute instance des types `int` et `float`.
- 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.
- Dans un système unix, 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")`.

2 Préliminaires

1. 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.
   solution :

```python
import numpy as np
from math import floor, ceil, log, sqrt
log(0.5)
```

2. 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.
   solution :

```python
def sont_proches(x,y):
    atol, rtol = 10**-5, 10**-8
    return abs(x-y) <= atol + abs(y) *rtol
```

3.
   solution : Cet appel renvoie 3.

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

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

   solution : On a :

$$\begin{aligned}\texttt{mystère}(1001, 10) &= 1 + \texttt{mystère}(100, 10)\\ &= 2 + \texttt{mystère}(10, 10)\\ &= 3 + \texttt{mystère}(1, 10)\\ &= 3\end{aligned}$$

5. Exprimer ce que renvoie `mystere` en fonction de la partie entière d'une fonction usuelle.

   solution : On constate que `mystère(x,b)` renvoie l'exposant maximal k tel que $b^k \leqslant n$.
   Cela revient à renvoyer l'entier k tel que $b^k \leqslant x < b^{k+1}$, c'est-à-dire $k \leqslant \log_b(x) < k+1$ c'est-à-dire $\log_b(x) - 1 < k \leqslant \log_b(x)$ c'est-à-dire $k = \lfloor \log_b x \rfloor$.
   Remarque : C'est le nombre de chiffres pour écrire n en base b moins 1.
   Remarque : Erreur d'énoncé : b doit être $\geqslant 2$ sans quoi la fonction peut ne pas terminer.

6. On donne le code suivant :

```python
pas = 1 -e5
x2 = 0
for i in range ( 1 0 0 0 0 0 ) :
    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.

solution : À la fin de la boucle `x1` vaut $10^5 \times 10^{-5}$ et `x2` vaut $\sum_{k=0}^{10^5-1} 10^{-5}$.

Si les calculs étaient exacts les deux contiendrait 1. Mais ce n'est pas le cas car les nombres manipulés sont des flottants. Notamment le calcul de `x2` utilise des additions entre des flottants d'ordre de grandeur très différent (0.99999 et 0.00001 pour la dernière), opération réputée pour être peu précise.

3 Génération de nombres premiers

3.1 Approche systématique

1.
solution : *Remarque :* Les tests que j'ai effectués semble montrer que Python utilise en réalité moins de 32 bits par cases pour enregistrer un tableau de booléens, mais bon, faisons comme si...

Une mémoire vive de 4Go, soit $4 \times 10^9 \times 8$ bits permet d'enregistrer un tableau de $\dfrac{4 \times 10^9 \times 8}{32}$ cases, c'est-à-dire 10^9 cases.
Remarque : Plusieurs élèves pensent que $1ko$ vaut $1024o$. En réalité, l'unité qui vaut 1024 octet est le « kibioctet », abrégé en « kio ». De même, $1Gio$ vaut 1024^2 octets.

2.
solution : Comme il n'y a que deux valeurs pour un booléen, on peut les coder par un seul bit. Ce qui permettrait un gain mémoire d'un facteur de presque 32 (presque car le coût du tableau lui-même reste le même).

3.
solution : *Remarque :* Il y a plusieurs difficultés techniques :
- L'élément i est dans la case $i-1$ du tableau.
- Boucle « pour » qui inclut les bornes : attention au passage au range.
- Lorsque N est trop grand, `int(sqrt(N))` ne marche pas car passage en flottant. Je n'ai cependant pas traité ce problème dans le corrigé.

```
def erath_iter(N):
    liste_bool=[True for i in range(N)]
    liste_bool[0]=False # 1 n'est pas premier
    for i in range(2, int(N**0.5)+1): #Attention à ne pas exclure racine de N dans le cas
        ↪ d'un carré parfait.
        if liste_bool[i-1]:
            for k in range(2*i-1, N, i):
                liste_bool[k]=False
    return liste_bool
```

4.
solution : Soit p un nombre premier. La ligne « Marquer comme faux les multiples de p différents de p » s'exécute en $O\left(\frac{N}{p}\right)$ (si elle est bien codée...)

La complexité totale est donc $\sum_{p \text{ premier} \leqslant N} O\left(\dfrac{N}{p}\right)$, qui vaut $O_{N\to\infty}\left(\sum_{p \text{ premier} \leqslant N} \dfrac{N}{p}\right)$ (série à termes positifs qui diverge), et donc $O_{N\to\infty}\left(N\log(\log(N))\right)$.

5.
solution : Soit b un base et n le nombre de chiffres de N. Alors $N = O(b^n)$. Si l'on veut faire des calculs avec les « grands O », il faudra utiliser des propriétés du genre « On peut passer au log dans des grand O si les suites sont strictement positives et tendent vers l'infini », propriétés pas spécialement citées dans les programmes officiels de maths et d'info. Je préfère utiliser de braves inégalités.
Ainsi, je pars du fait que $N < b^n$ (car le plus grand entiers qu'on peut écrire avec n chiffres en base b a tous ses chiffres égaux à $b-1$, c'est donc $\sum_{i=0}^{n-1}(b-1)b^i$, soit $(b-1)\frac{1-b^n}{1-b} = b^n - 1$.)

J'obtiens alors que $N\log(\log(N)) < b^n \log(n \log b) = O_{n\to\infty}(b^n \log n)$.

4 Génération rapide de nombres premiers

1.

solution : L'énoncé est particulièrement pas clair, mais j'ai cru comprendre qu'on prenait $x_0 = 0$. On aura $A = \sum_{i=1}^{N-1} 2^i = 2^N - 2$.

2.

solution : Remarque : L'algo revient juste à tirer au (pseudo-)hasard chaque bit du nombre à construire, en commençant par les poids faibles. En outre la suite à utiliser pour générer des nombres pseudo-aléatoires est précisée.

```python
def bbs(N):
    p1 = 24375763
    p2 = 28972763
    M=p1*p2
    h = time.time()
    xi=floor((h-floor(h))*10**7 )# Partie fractionnaire de h. La résolution de l'horloge est 10**-7 d'après l'énoncé.
    A=0
    for i in range(N):
        if xi%2==1:
            A = A+2**i
        # Rema : pour condenser les deux lignes ci-dessus en une seule :
        # A += (xi%2)*2**i
        # Autre remarque : pas de contrainte de complexité dans l'énoncé, mais il serait plus efficace de calculer les puissances de deux au fur et à mesure.
        xi=(xi**2)%M
    return A
# Dans mes tests, bbs(4) ne renvoie jamais 13. Pourquoi ??
# Réponse d'Éric Detrez :
"""
Le problème est que, si on suit l'énoncé, la graine est la partie fractionnaire de time() donnée comme calculée avec 7 décimales. On calcule donc une graine avec 7 chiffres. Mais alors x1 < M (produit de deux premiers sur 8 chiffres) donc x0 et x1 ont la même parité.
Comme le signale le collègue qui écrit le contrôle de l'épreuve : on ne trouve jamais 13. En fait tous les entiers trouvés sont congrus à 0 ou 3 modulo 4.

Comme l'énoncé est *très* mal posé, on peut peut être comprendre qu'il semble suggérer de commencer avec x1 (ce qui ne donnerait que des entiers pairs ...). Bref, la bouillie habituelle de cette épreuve.
"""
```

3.

solution : Pour `premier_rapide` : la question antéprécédente montre que pour tout $N \in \mathbb{N}$, l'appel `bbs(N)` renvoie un entier aléatoire entre 0 et $2^N - 1$. Ce n'est pas très pratique ici car `nb_max` n'a selon l'énoncé aucune raison d'être une puissance de deux.

Cependant, `mystere(nb_max, 2)` permet de récupérer le plus grand entier N tel que $2^N \leqslant \texttt{nb\_max}$. Ainsi, `bbs(mystere(nb_max, 2))` renvoie bien un nombre entier strictement inférieur à N. Mais tous les entiers entre 2^N et $\texttt{nb\_max} - 1$ seront ignorés.

Le sujet aurait peut-être mieux fait d'imposer que `nb_max` soit une puissance de deux, ou d'utiliser le nombre de bits comme paramètre.

Aure difficulté : `bbs` pourrait renvoyer 0 comme valeur de `p` auquel cas le calcul de `a**(p-1)` échoue, ou alors 1 auquel cas `a**(p-1)%p` vaut 1 sant que $p$soit premier. Il faut donc éliminer ces deux cas.

Enfin, le test de Fermat ne fonctionne pas pour $a = p$ (d'où la condition $a > p$ dans l'énoncé). En effet, $p^{p-1} \equiv 0 \not\equiv 1 [p]$. Ainsi par exemple le nombre 7 sera jugé non premier par ce test et sera donc écarté. C'est la raison pour laquelle l'énoncé demande que $\texttt{nb\_max} \geqslant 12$: ainsi il y aura au moins un nombre premier qui soit > 7 et $< \texttt{nb\_max}$ (c'est 11).

Mais alors nouvelle difficulté : par exemple pour $\texttt{nb\_max} = 12$, on aura $\texttt{mystere}(\texttt{nb}_\texttt{m}\texttt{ax}, 2) = 3$ donc les nombres seront tirés au hasard par `bbs` dans $[\![0, 7]\!]$, ils seront alors tous jugés non premier par notre algo. Au final il faudra prendre $\texttt{nb\_max} \geqslant 16$ pour que $\texttt{mystere}(\texttt{nb}_\texttt{m}\texttt{ax}, 2)$ renvoie 4 et que l'algo puisse terminer.

```python
def a_l_air_premier(p):
    """ Entrée : p > 1
```

```
        Sortie : un booléen qui indique si p à l'air premier... Précilément :
            - si p n'est pas premier ou p <= 7 alors le booléen renvoyé est Faux
            - si p est premier et p > 7 le booléen est Vrai avec une forte probabilité...
            - Cette fonction n'est pas fiable pour p €[2,5,7]. On peut la corriger " à la main " mais ça serait moche."""
    res=True
    for a in [2,3,5,7]:
        if a**(p-1) % p != 1 : res=False
    return res

def premier_rapide(n_max):
    N = mystere(n_max, 2)
    p = bbs(N)
    while p<2 or not(a_l_air_premier(p)) :
        p=bbs(nb_chiffres)
    return p
```

4.
solution :

```
€
def stats_bbs_fermat(N, nb):
    """ Prendre N>15 sinon ça plante à cause de premier_rapide(N)."""
    premier = [False]+erath_iter(N) #Je décale la liste pour que ça soit plus pratique. (Ce n'est certainement pas ce que souhaitait le correcteur.)

    erreurs=[]
    for k in range(nb):
        p=premier_rapide(N)
        if not premier[p]:
            erreurs.append(p)
    return erreurs, len(erreurs)/nb
```

5 Compter les nombres premiers

5.1 Calcul de (n) via un crible

1.
solution :

```
€
def Pi(N):
    premier = [False]+erath_iter(N)
    res=[]
    pin=0
    for n in range(1,N+1):
        if premier[n]:
            pin+=1
        res.append([n, pin])
    return res
```

2.
solution :

```
€
def verif_Pi(N):
    valeurs_de_pi=Pi(N)[5392:]
    res=True
    for (n, pin) in valeurs_de_pi:
        if n/(log(n)-1)  >= valeurs_de_pi[n][1]:
            res=False
    return res
```

5.2 Calcul approché via une intégrale généralisée

5.2.1 Estimation de li par quadrature numérique

1.
solution : Nous voudrons obtenir des résultats les plus précis possibles, ce qui sera obtenu lorsque `pas`tend vers 0. Ainsi, il est pertinent d'exprimer la complexité en fonction de `pas`, lorsque celui-ci tend vers 0. En outre, nous voudrons également prendre x grand, car pour des petiter valeurs, la valeur exacte de $\pi(n)$ peut être calculée. En résumé, nous allons exprimer la complexité en fonction de x et `pas`, lorsque le premier tend vers ∞ et le second vers 0.

   L'opération élémentaire la plus couteuse est sans doute l'appel à la fonction à intégrer, qui est en $O(1)$ d'après l'énoncé. Il y a un tel appel pour chaque rectangle de la subdivision, et il y a $\frac{x}{\text{pas}}$ rectangles. D'où une conplexité en $O\left(\frac{x}{\text{pas}}\right)$.

2.
solution :

```python
€
def inv_ln_rect_d(a,b,pas):
    nb_pas=int(round( (b-a)/pas )) # Attention aux erreurs d'arrondi : utiliser round et non pas int.
    # J'ai du rajouter ensuite un int pour les tracés (quand a et b viennent d'un linspace, round ne renvoie plus un entier...)
    somme=0
    for k in range(1, nb_pas+1):
        somme += 1/log(a+k*pas)
    return somme*pas
```

3.
solution :

```python
€
def li_d(x,pas):
    print(x)
    # Cohérent avec la figure.
    # Crée une erreur de 1.
    if abs(1-x)<pas:
        return -float("infinity")
    elif x>1:
        return inv_ln_rect_d(0, 1-pas, pas) + inv_ln_rect_d(1+pas,x, pas)
    else:
        return inv_ln_rect_d(0,x,pas)
```

4.
solution : Les méthodes des rectangles centrés et des trapèzes ont la même complexité. Toutefois la méthode des trapèzes fournit un résultat plus précis si la fonction est de classe C^2. La méthode des rectangles centrés est hors programme.

5.2.2 Analyse des résultats de `li_d`

1.
solution : L'écart relatif présente une asymptote en une valeur proche de 1.4 tout simplement car son calcul fait intervenir une division par `li_ref` qui s'annule (visible sur la figure 1).

2.
solution : Voici mon hypothèse et quelques commentaires concernant cette question :

   - Tout d'abord, à partir du moment où on n'est pas en train d'intégrer une fonction continue (par morceaux) sur un segment, il n'y a aucune raison que la méthode des rectangles converge.
     Je précise d'ailleurs qu'aucun théorème précis de convergence n'est cité au programme, je précise également que la notion d'intégrale impropre où la singularité est à l'intérieur de l'intervalle d'intégration n'est pas au programme de math, même de MP.
   - Du fait que $\ln(1-\epsilon)$ et $-\ln(1+\epsilon)$ sont égaux au premier ordre lorsque $\epsilon \to 0$, on déduit que $\lim_{\epsilon\to 0} \int_{1-\epsilon}^{1+\epsilon} \frac{1}{\ln(1+t)}\,\mathrm{d}t = 0$ (le calcul est laissé au lecteur, utiliser Taylor-Lagrange).
     Plus précisément, $\displaystyle\int_{1-\epsilon}^{1+\epsilon} \frac{1}{\ln(1+t)}\,\mathrm{d}t = O_{\epsilon\to 0}(\epsilon)$.
   - Voici alors une manière de corriger le calcul. On commence par fixer ϵ. Les intervalles $[0, 1-\epsilon]$ et $[1+\epsilon, x]$ sont de braves segments sur lesquels la fonction $x \mapsto \frac{1}{\ln x}$ est définie et continue. Donc la méthode des rectangles converge sur

ces intervalles, et il existe un pas tel que l'erreur soit inférieure à ϵ. Comme en outre, l'erreur commise en supprimant $[1-\epsilon, 1+\epsilon]$ de l'intervalle d'intégration est aussi de l'ordre de ϵ, nous aurons une erreur totale en $O(\epsilon)$.
Donc en supprimant la condition `pas` $= \epsilon$, nous pouvons obtenir un résultat correct. Dans le corrigé, j'ai pris un pas égal à ϵ^2, et le résultat semble correct.

```
€
def li_d_pas_differents(x,eps):
    """ Proposition de correction de li_d. On prend pas=eps**2."""
    if abs(1-x)<eps:
        return float("infinity")
    elif x>1:
        return inv_ln_rect_d(0, 1-eps, eps**2) + inv_ln_rect_d(1+eps,x, eps**2)

    else:
        return inv_ln_rect_d(0,x,eps**2)
```

5.3 Estimation de li via Ei

1.

solution : Pour obtenir une complexité en $O($`MAXIT`$)$, il faut penser à calculer les factorielles et les puissances de x en les gardant en mémoire au fur et à mesure.
L'écriture propre des invariants de boucle peut vous sauver dans ce genre de fonction...

```
€
MAXIT=100 # L'énoncé semble sous-entendre que MAXIT est une variable globale

def Ei(x):
    if x<=0:
        return False
    else :
        gamma=0.577215664901

        n=1
        n_fact=1 #n!
        xpn=x #x**n
        Ein_moins_un = gamma+log(x)

        Ein = Ein_moins_un + x



        while n <= MAXIT and not sont_proches(Ein, Ein_moins_un):
            #Invariant de boucle : Ein_moins_un contient Ei_(n-1) et Ein contient Ei_n
            #                      n_fact contient n!
            #                      xpn contient x**n
            n+=1
            n_fact*=n
            xpn*=x

            Ein_moins_un=Ein
            Ein += xpn/(n * n_fact )
        if n==MAXIT+1:
            return False
        else:
            return Ein

def li_dev(x):
    return Ei(log(x))
```

6 Évaluation des performances (BDD)

1. Plusieurs enregistrements ont la même valeur pour le champ `nom`.

2. (a) Nombre d'ordinateurs disponibles et quantité moyenne de mémoire vive.

```sql
SELECT COUNT(*) AS nb_ordi, AVG(RAM) AS RAM_moyenne
      FROM ordinateurs
```

(b) Noms des PC sur lesquels l'algorithme `rectangles` n'a pas été testé pour la fonction `li`.

```sql
SELECT nom FROM ordinateurs
  WHERE nom NOT IN (
    SELECT teste_sur FROM fonctions
    WHERE algorithme="rectangles" and nom = "li"
  )
```

Ou alors utiliser **EXCEPT** :

```sql
SELECT nom FROM ordinateurs
EXCEPT
SELECT teste_sur FROM fonction
      WHERE  algorithme="rectangles" and nom = "li"
```

(c) Pour chaque test de `Ei` garder le nom de l'algo, du pc, et sa puissance. Trier du plus lent au plus rapide.

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