MINES-PONTS - 2019 - INFORMATIQUE COMMUNE

Rédigé par Jérémy Larochette (Lycée Carnot, Dijon) et Laurent Sartre (Lycée Montaigne, Bordeaux).

Partie I. Préliminaires

❏ **Q1** –

```
from math import log, sqrt, floor, ceil

print(log(0.5))
```

❏ **Q2** –

```
def sont_proches(x, y):
    "test si |x - y| <= 1e-5 + |y| 1e-8"
    atol = 1e-5
    rtol = 1e-8
    return abs(x - y) <= atol + abs(y) * rtol
```

❏ **Q3** – L'appel de `mystere(1001, 10)` renvoie `3`.
Plus généralement, `mystere(a, b)` renvoie le nombre de chiffres de l'écriture en base `b` de `a` moins 1.

❏ **Q4** – Si $b = 1$, la fonction boucle indéfiniment. Sinon, si $b^n \leqslant x < b^{n+1}$, la fonction `mystere` va effectuer des appels jusqu'à avoir divisé n fois, car on a $1 \leqslant \frac{x}{b^n} < b$ et la fonction s'arrête. Elle renvoie donc la valeur $\boxed{n = \lfloor \log_b a \rfloor}$.

❏ **Q5** – À la fin de l'exécution `x1` contient le résultat de la multiplication de 100000 par 10^{-5} et `x2` le résultat des 100000 additions de 10^{-5} à partir de 0.

Le codage des nombres flottants par la norme IEEE 754 provoque une perte de précision plus importante lors des additions (répétées, ici, un grand nombre de fois) que lors de la multiplication. De plus, 10^{-5} est codé de manière approchée.

Remarque : L'erreur de représentation de 10^{-5} *sur* 32 *bits est de l'ordre de* 10^{-13} *ce qui correspond à l'ordre de grandeur de l'erreur obtenue sur* `x2`.

Partie II. Génération des nombres premiers

II.a Approche systématique

❏ **Q6** – Comme 32 bits représente 4 octets, on pourra approximativement stocker, dans 4 Go, $\frac{4 \cdot 10^9}{4} = 10^9$ booléens en utilisant les données de l'énoncé.

❏ **Q7** – Un booléen pouvant être codé sur un seul bit, on peut gagner un facteur 32.

❏ **Q8** –

```
def erato_iter(N):
    liste_bool = [True for _ in range(N)]
    liste_bool[0] = False
    i = 2
    # /*\ Un for i in range(2, floor(sqrt(N)) + 1 pourrait être incorrect pour N grand
    # à cause de l'approximation des flottants.
    while i ** 2 <= N:
        if liste_bool[i - 1]: # Si i est premier
            m = i * 2 # premier multiple non encore rencontré
            while m <= N:
                liste_bool[m - 1] = False # m n'est pas premier
                m += i
        i += 1
    return liste_bool
```

❏ **Q9** – En majorant le nombre de tours de la boucle interne exécutée pour les `i`$= p < N$ premiers par le nombre $\left\lfloor \frac{N}{p} \right\rfloor$ de multiples de p dans $[\![1, N]\!]$, la complexité de la fonction précédente est

$$O\left(\sum_{p<\sqrt{N}, p \text{ premier}} \left\lfloor \frac{N}{p} \right\rfloor \right) = O\left(\sum_{p<\sqrt{N}, p \text{ premier}} \frac{N}{p} \right) = O\left(N \sum_{p<\sqrt{N}, p\text{premier}} \frac{1}{p} \right) = O(N \ln(\ln \sqrt{N})) = O(N \ln(\ln N))$$

car d'après l'énoncé, $\displaystyle\sum_{p<N, p \text{ premier}} \frac{1}{p} \sim \ln(\ln N)$.[1]

❏ **Q10** – Il est naturel de s'intéresser à la représentation informatique de N et donc du nombre n de chiffres de l'écriture en base 2 de N. Comme $n = \lfloor \log_2(N) \rfloor + 1 \sim \log_2(N) = \frac{\ln N}{\ln 2}$, la complexité est alors $O(2^n \ln(n))$.

II.b Génération rapide de nombres premiers

❏ **Q11** – Si x_i est impair à chaque itération, A est l'entier codé en binaire par N bits à 1, soit $A = 2^N - 1$.

❏ **Q12** –

```
from time import time

def bbs(N):
    p1 = 24375763
    p2 = 28972763
    M = p1 * p2
    # calculer la graine
    t = time()
    xi = int((t - int(t)) * 1e7)
    A = 0
    for i in range(N):
        if xi % 2 == 1: # si xi est impair
            A = A + 2**i
        # calculer le nouvel xi
        xi = (xi ** 2) % M
    return A
```

1. Ce résultat peut s'obtenir à partir du (difficile) théorème des nombres premiers qui dit que $\pi(n) \sim \frac{n}{\ln n}$. On en déduit que le n^e nombre premier p_n vérifie $p_n \sim n \ln n$, puis par une comparaison à une intégrale (série de Bertrand) et par comparaison de sommes partielles divergentes, on obtient le résultat.

❑ **Q13** – L'entier renvoyé par `bbs` est au plus égal à $2^N - 1$. Il faut donc que $2^N - 1 < n_{max}$ donc que $2^N \leqslant n_{max}$ soit $N \leqslant \log_2 n_{max}$.

```
def premier_rapide(n_max):
    "Renvoie un nombre probablement premier au plus égal à n_max."
    N = floor(log(n_max, 2))
    while True:
        p = bbs(N)
        if p > 8:
            bool = True
            for a in (2, 3, 5, 7):
                bool = bool and (pow(a, p - 1, p) == 1)
                # a ** (p - 1) % p est moins optimisé
            if bool:
                return p
```

Ou, récursivement,

```
def premier_rapide(n_max):
    "Renvoie un nombre probablement premier strictement inférieur à n_max."
    N = floor(log(n_max, 2))
    p = bbs(N)
    if p <= 8:
        return premier_rapide(n_max)
    for a in (2, 3, 5, 7):
        if pow(a, p - 1, p) != 1
            return premier_rapide(n_max)
    return p
```

❑ **Q14** –

```
def stats_bbs_fermat(N, nb):
    """renvoie la proportion et la liste des faux-positifs dans la génération de nb
nombre premiers"""
    liste_bool = erato_iter(N)
    liste = []
    for _ in range(nb):
        p = premier_rapide(N + 1) # p <= N
        if not liste_bool[p - 1]: # si p n'est pas premier
            liste.append(p)
    return (len(liste) / nb, liste)
```

Partie III. Compter les nombres premiers

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

❑ **Q15** –

```
def pi(N):
    "renvoie une liste de listes [n, pi(n)] pour n entre 1 et N."
    liste_bool = erato_iter(N)
    liste = []
    pi_n = 0
    for n in range(1, N + 1):
        if liste_bool[n - 1]:
            pi_n += 1
        liste.append([n, pi_n])
    return liste
```


❑ **Q16** –

```
def verif_Pi(N):
    "teste si n / (ln(n) - 1) < pi(n) pour tout n entre 5393 et N."
    def f(n):
        return n / (log(n) - 1)
    liste = pi(N)
    for n, pi_n in liste[5392:]:
        if f(n) >= pi_n:
            return False
    return True
```

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

❑ **Q17** – La méthode des rectangles à droite consiste à effectuer un nombre fixe d'opérations élémentaires (évaluation de f, somme) pour chaque rectangle. Si on appelle n le nombre de rectangles, on obtiendra donc $O(n)$.

Remarque : si on intègre sur $[a, b]$ avec un pas de h, n est du même ordre que $\frac{b-a}{h}$ donc que $\frac{1}{h}$, ce qui fait une complexité en $O\left(\frac{1}{h}\right)$.

❑ **Q18** – La complexité des autres algorithmes est aussi $O(n)$.

❑ **Q19** – Vu les considérations de la question **Q5**, on évite d'ajouter un grand nombre de fois le pas à une variable `x`.

```
def inv_ln_rect_d(a, b, pas):
    """Calcule une valeur approchée de l'intégrale entre a et b de 1/ln par la méthode
des rectangles à droite, avec 1 n'appartenant pas à [a, b]."""
    I = 0
    x = a + pas
    n = 1
    while x <= b:
        I += 1/log(x)
        n += 1
        x = a + n * pas
    return I * pas
```

❑ **Q20** –

```
def li_d(x, pas):
    if x == 1:
        return -float('inf')
    if x < 1:
        # x et 1 sont multiples de pas donc x <= 1 - pas
        return inv_ln_rect_d(0, x, pas)
    return inv_ln_rect_d(0, 1 - pas, pas) + inv_ln_rect_d(1 + pas, x, pas)
```

❑ **Q21** – Comme, au voisinage de 1,4, $\mathrm{li}(x)$ est proche de 0 et comme l'écart absolu est non nul, l'écart relatif tend vers l'infini.

❑ **Q22** – Comme, au voisinage de 1, $\frac{1}{\ln(1+\varepsilon)} \sim \frac{-1}{\ln(1-\varepsilon)}$, c'est-à-dire que le point de coordonnées $(1,0)$ est (approximativement) centre de symétrie pour $\frac{1}{\ln}$, on s'attend à ce que, pour ε suffisamment petit, l'intégrale de $1-\varepsilon$ à $1+\varepsilon$ soit nulle.

Or, en observant la figure 6, on se rend compte que, lorsque l'on est près de 1, les rectangles (à droite) de l'approximation de la partie négative cette intégrale sont, par symétrie, compensés par ceux de la partie positive, sauf celui de $1-2\varepsilon$ à $1-\varepsilon$. Comme l'aire de ce rectangle est approximativement égale à $\varepsilon\cdot\dfrac{1}{\varepsilon}=1$, on obtient bien un écart absolu de l'ordre de 1 dans l'approximation de li par la méthode des rectangles à droite.

❏ **Q23** – Il suffit donc, pour $x>1$, d'ajouter 1 à la valeur calculée.

On peut aussi, plus proprement, pour conserver la symétrie, appliquer les rectangles à droite à gauche de 1 et les rectangles à gauche à droite de 1.

❏ **Q24** –

```
MAXIT = 100

def EI(x):
    gamma = 0.577215664901
    k = 1
    fact_k = 1 # factorielle de k
    x_puiss_k = x # x puissance k
    ei = gamma + log(x)
    ei1 = ei + x / (k * fact_k)
    while k <= MAXIT and not sont_proches(ei, ei1):
        k += 1
        fact_k *= k
        x_puiss_k *= x
        ei = ei1
        ei1 = ei + x_puiss_k / (k * fact_k)
    if k == MAXIT + 1:
        return False
    return ei1

def li_dev(x):
    if x <= 1:
        return False
    log_x = log(x)
    if log_x > 40:
        return False
    return Ei(log_x)
```

Partie IV. Analyse de performance de code

❏ **Q25** – Il n'est pas possible d'utiliser `nom` comme clé primaire car il ne vérifie pas la contrainte d'unicité pour chaque tuple de la table `fonctions`. Il est préférable d'utiliser `id` par exemple.

❏ **Q26** –

1.

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

2.

```
SELECT nom FROM ordinateurs
EXCEPT
SELECT teste_sur FROM fonctions
WHERE nom = "li" AND algorithme = "rectangles";
```

ou

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


ou

```
SELECT nom FROM ordinateurs o
WHERE NOT EXISTS (SELECT * FROM fonctions
                  WHERE teste_sur = o.nom AND nom = "li"
                                          AND algorithme = "rectangles");
```

ou

```
SELECT nom FROM ordinateurs o
WHERE 0 = (SELECT COUNT(*) FROM fonctions
                  WHERE teste_sur = o.nom AND nom = "li"
                                          AND algorithme = "rectangles");
```

3.

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