Nom : .....

Prénom : .....

2023/12/06

Classe : MP/PSI

Lycée Loges

ITC : DS n° 2 - 2 heures
calculatrice interdite

Exercice I : ···· /-

Note finale : ···· / -

⚠ Faites attention à la lisibilité de votre code et vos explications

Exercice I

Introduction

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 clé 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`. 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 traiter en langage `SQL`.

Définition, rappels et notations

- Un nombre premier est un entier naturel qui admet exactement deux diviseurs : 1 et lui-même. Ainsi, 1 n'est pas un nombre 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)` envoie 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 $n \leq x < n+1$.
- `round(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 `times()` du module `time` renvoie un flottant représentant le nombre de secondes depuis le premier janvier 1 970 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.

Partie I

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.

2. Écrire une fonction `sont_proche(x,y)` qui renvoie `True` si la condition suivante est remplie et `False` sinon :

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

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.

3. 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)
```

a) Que se passe-t-il lorsque $b = 1$?

b) On suppose que $b^n \leq x < b^{n+1}$, où $n \in \mathbb{N}$. Combien d'appels récursifs y a-t-il lors de l'exécution de `mystere(x,b)` et que renvoie cette fonction ?

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

4. On donne le code suivant :

```python
pas = 1e-5
x2 = 0
for i in range(100000):
    x1 = (i+1) * pas
    x2 = x2 + pas
print("x_1:", x1)
print("x2:", x2)
```

L'exécution de ce code produit le résultat suivant :

```
x1: 1.0
x_2: 0.9999999999980838
```

Commenter le résultat obtenu.

Partie II

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 :

```
DONNEES : N, entier superieur a 1
RESULTAT : liste_bool, liste de booleens
DEBUT
    liste_bool <-  liste de N booleens initialises a Vrai
    Marquer comme Faux le premier element de liste_bool
    POUR entier i <- 2 a partie entiere de racine carre de N
        FAIRE
        SI i n'est pas marque comme Faux dans liste_bool
           ALORS  Marquer comme faux les multiples de i
                  differents de i dans liste_bool
        FIN SI
    FIN POUR
    RETOURNER liste_bool
FIN
```

À la fin de l'exécution, `liste_bool[i]` vaut `True` si $i+1$ est premier. Par exemple, pour `N=4` une implémentation `python` du crible renvoie `[False,True,True,False]`.

5. Sachant que le langage `python` traite les listes de booléens comme des listes 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 ?

6. Quel facteur peut-on gagner sur la valeur maximal 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 donnée (préciser cet espace mémoire) ?

7. Écrire la fonction `erato_iter(N)` qui implémente l'algorithme du crible d'Ératosthène pour un paramètre N qui est supérieur ou égal à 1.

8. Démontrer que la complexité du crible d'Ératosthène est en $O(\ln(\ln(N)))$.

9. Que vaut le nombre de chiffres en base 2, noté n, de N. En déduire la compléxité de l'algorithme en fonction de n.

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 unes au hasard. Une meilleur 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 :

— générer un entier pseudo-aléatoire (voir ci-dessous)

— vérifier si cet entier a de forts chances d'être premier

— 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 :

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

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.

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

11. Compléter la fonction `bbs(N)` donnée ci-dessous qui réalise ces itérations. La graine est un entier représentant la fraction de seconde du temps courant, par exemple 1528287738.7931523 donne la graine 7931523. Le paramètre N est un entier non nul.

```
... (A completer)
def bbs(N):
    p1 = 24375763
    p2 = 28972763
    M = p1 * p2
    # calcul de la graine
    ... (a completer)
    xi = ... (a completer)
    A = 0
    for i in range(N):
        if  ... (a completer)
            A = A + 2**i
        # calculer le nouvel xi
        xi = ... (a completer)
    return A
```

Le test 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 que p soit premier.

12. 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.

13. On souhaite caractériser le taux d'erreur 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`, s'ils sont réellement premiers. Cette fonction renvoie le taux relatif

d'erreur ainsi que la liste des faux nombres premiers trouvées. Les paramètres `N` et `nb` sont des entiers strictement positifs.

Partie III

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

14. Écrire une fonction `Pi(N)` qui calcule la valeur de $\pi(n)$ pour tout entier n de $[\![1,n]\!]$. Les nombres premiers sont déduit de la liste `liste_bool` renvoyée par la fonction `erato_iter` de la question 7. 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 $n \geq 5393$. On souhaite vérifier cette inégalité en se basant sur la fonction `Pi(N)` de la question 14.

15. 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 supérieur ou égal à 5 393.

Partie IV

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 `ordinateur` et permet de stocker des informations sur les ordinateurs utilisés pour les tests. Ses attributs sont :

- `nom CHAR`, 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 stoke les informatioons sur les tests effectués pour différentes fonctions en cours de développement. Ses attributs sont :

- `id INTEGER`, l'identifiant du test effectué.
- `nom CHAR`, le nom de la fonction testée.
- `algorithme CHAR`, le nom de l'algorithme qui permet le calcul de la fonction testée.
- `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	rectangle	nyarlathotep165	2638
2	li	rectangle	shubniggurath28	736
3	li	trapezes	nyarlathotep165	4842
...				
2154	Ei	puiseux	nyarlathotep145	2766
2155	aleatoire	bbs	azathoth145	524

16. Expliquer pourquoi il n'est pas possible d'utiliser l'attribut `nom` comme clé primaire de la table `fonction`.

Écrire les requêtes SQL permettant d'obtenir les tables de données suivantes :

17. Donner les noms des PC ayant une mémoire vive de 32 Go.

18. Donner les noms des PC et les noms des fonctions qui y sont testées.

19. Donner la liste des algorithmes permettant le calcul des fonctions testées. On ne souhaite pas avoir de doublons dans les données obtenues.

20. Donner le nom des ordinateurs ayant un temps d'exécution supérieur strictement à 1000 millisecondes pour la fonction `li`.

21. 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.

22. Donner les noms des PC sur lesquels l'algorithme `rectangles` n'a pas été testé pour la fonction nommée `li`

23. Combien de PC ont été testé avec la fonction `li` ?

24. Donner les noms des fonctions ainsi que la moyenne de leurs temps d'exécution.

25. Donner les noms des PC dont le temps d'exécutions pour la fonction `li` est supérieur à la moyenne du temps d'exécution de cette fonction.

26. Quel résultat obtient-on avec la requête suivante ?

```
SELECT teste_sur
FROM fonctions
GROUP BY teste_sur
HAVING COUNT(id) =
        SELECT MAX(nbr)
        FROM SELECT COUNT(id) AS nbr
                    FROM fonction
                    GROUP BY teste_sur
```