TSI2. Concours Blanc 2021

Épreuve d'informatique

Duré 3 heures. Les calculatrices sont autorisées

Utilisation des nombres premiers pour le codage

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 clès 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 leur produit, il est très difficile de déduire les deux facteurs premiers.

Les programmes demandés ici sont à rédiger en PYTHON.

Définitions, rappels, 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.
- 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 $\lfloor x \rfloor$, 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.
- $log(x, a)$ renvoie sous forme de flottant la valeur du logarithme de x en base a.

On rappelle que si $x > 0$, $log(x, a) = \dfrac{log(x)}{log(a)}$. Par voie de conséquence, $a > 0$ et $a \neq 1$.

- 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).

bullet 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 est : $float("inf")$
- En PYTHON 3, on peut utiliser des entiers illimités de plus de 32 bits avec le type *long*
- 10^{-3} par exemple s'écrit en PYTHON $1e-3$
- Dans une fonction booléenne, si l'on veut tester une condition qui est par exemple $a < b$ ou $a <= b$ ou $a = b$ etc. alors on écrit *return condition* et la fonction renvoie $True$ si la condition est vérifiée et $False$ sinon.
- La syntaxe pour remplir une liste L de $N+1$ quantités a est :

$L = [a,\ for\ i\ in\ range(N+1)]$

- On rappelle que $L[1:]$ fournit la sous-liste issue de L en enlevant $L[0]$.
- $x \,\%\, 2$ donne le reste de la division euclidienne de x par 2 et donc $x \,\%\, 2 == 1$ signifie que x est impair.
- En important *numpy as np*, on rappelle que $np.arange(a, b, h)$ crée la liste des (a_k) avec $a_0 = a$, $a_1 = a + h$, etc. et $a_N = a + Nh = a + N\frac{b-a}{N} = b$.

Partie I. Préliminaires

Q01 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.

Remarque : on peut importer toutes les fonctions du module ou que les précitées. Au choix.

Q02 Écrire une fonction booléenne $sont\_proches(x, y)$ 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 la fonction et valant respectivement 10^{-5} et 10^{-8}. Les arguments x et y sont des réels quelconques.

Q03 On donne la fonction $mystere$ ci-dessous. Ici x est un réel strictement positif et b un entier supérieur ou égal à 2.

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

Que renvoie $mystere(1001, 10)$?
Que se passe t-il si $b = 1$?

Q04 Exprimer ce que renvoie $mystere$ en fonction de la partie entière d'une fonction usuelle.

Indication : si $x \geqslant b$, on remarquera que l'entier renvoyé par $mystere(x, b)$ est le plus petit entier $k \geqslant 1$ tel que $x/b^k < b$. Et donc $b \leqslant x/b^{k-1}$. On écrira k en fonction du logarithme de base b.

Q05 On donne le code suivant :

```
>>> 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'Eratosthène est un algorithme (nommé **Algorithme 1**) 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
$liste\_bool \leftarrow$ 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

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 de cet Algorithme 1 renvoie $[False\ True\ True\ False]$

Q06 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 d'octets ?

Remarque : on rappelle qu'un octet vaut 8 bits.

Q07 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) ?

Q08 Justifier que si i et k sont des entiers non nuls, $\left\{ \begin{array}{l} 2 \leqslant k \text{ et} \\ i\,k \leqslant N \end{array} \right. \Leftrightarrow \left\{ \begin{array}{l} 2 \leqslant k \text{ et} \\ k \leqslant \lfloor \frac{N}{i} \rfloor \end{array} \right.$

Écrire alors en PYTHON la fonction $erato\_iter(N)$ qui implémente l'algorithme 1 pour un argument N qui est un entier supérieur ou égal à 1.

Remarque : l'indexation des listes commençant à l'indice 0, la case d'indice i représente en réalité l'entier $i+1$. Pour éviter ce problème de décalage d'indice, nous commençons par créer une liste de taille $N+1$ (la case d'indice i représente l'entier i), ce qui permet de suivre exactement l'algorithme 1. En sortie de boucle, notre liste de booléens contient une case de trop (celle correspondant à $i = 0$) et on renvoie la sous-liste privée de cet élément.

Q09 Montrer que $\lfloor \sqrt{N} \rfloor \sim \sqrt{N}$ quand N tend vers $+\infty$.

Déterminer la complexité de $erato\_iter(N)$ en fonction de N.

Remarque : On écrira cette complexité sous forme d'un $O(g(N))$, où g est une fonction de N à préciser.

On admettra que : $\displaystyle\sum_{p<N,\, p\, premier} \frac{1}{p} \equiv \ln(\ln(N))$.

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 p.
Donner une approximation du résultat de la question précédente en fonction de p.

Remarque : On rappelle que le nombre $a_{p-1}a_{p_2}...a_0$ s'écrit aussi
$a_{p-1}10^{p-1} + a_{p-2}10^{p-2} + ... + a_1 10^1 + a_0 10^0$.
On a pour tout entier i, $a_i \in [\![0, 9]\!]$ avec $a_{p-1} \neq 0$.
Montrer que N est compris entre 10^{p-1} et 10^p.

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 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 ($i \geqslant 1$) on calcule :

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

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 ?
En déduire que le nombre aléatoire généré par $bbs(N)$ est toujours strictement inférieur à 2^N.

Remarque : $\displaystyle\sum_{i=1}^{N-1} q^i = q\frac{q^{N-1}-1}{q-1}$, où $q \neq 1$.

Q12 Compléter avec le nombre de commandes que vous jugerez nécessaire sur les lignes ayant des points d'interrogation dans le code suivant qui construit la fonction $bbs(N)$ 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.

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

Remarque : Attention, pour la graine, on a à récupérer que les 7 chiffres de la partie décimale. Et il faut en plus convertir le flottant obtenu en entier.

Le test de primalité le plus simple est celui 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 Écrire une fonction booléenne $test\_fermat(L, p)$ qui teste si p entier vérifie le test de primalité de Fermat pour tous les éléments a appartenant à une liste L d'entiers donnés, c'est-à-dire si le reste de la division euclidienne de a^{p-1} par p vaut 1 ou non. On renvoie $False$ si ce n'est pas le cas. On renverra d'emblée $False$ dans le cas où $p = 0$ et $p = 1$.

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.

Remarque : pour construire $premier\_rapide$, on remarque qu'appelée avec l'entier N, la fonction bbs renvoie un entier inférieur ou égal à $2^N - 1$. Comme on veut un entier inférieur ou égal à n_max, il suffit de l'appeler avec le plus grand entier N vérifiant $2^N - 1 \leqslant n\_max$. On pourra ainsi affecter N à partir de la fonction *mystère* ou avec $floor$ et log. Puis on affecte la liste L constituée des entiers possibles pour a. Tant que l'entier (qu'on peut appeler A) obtenu avec la fonction bbs n'est pas probablement premier (ce que l'on pourra traduire en syntaxe PYTHON par $while\ not\ test\_fermat(L, A)$:) , on en tire un autre avec bbs. On retournera le dernier A (qui peut très bien être le premier affecté avant la boucle conditionnelle).

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.

Remarque : pour construire $stats\_bbs\_fermat(n, nb)$, on doit commencer par utiliser une fonction booléene qui renvoie $True$ si N est premier et $False$ sinon. Le plus simple est d'utiliser $erato\_iter(N)$ que l'on affecte dans une variable $premiers$. On affecte ensuite $nbfaux$ à $zéro$ et la liste $faux$ des faux nombres premiers à la liste vide. Puis dans une boucle, on génére nb nombres par la fonction $premier\_rapide(N)$ et on teste si ces nombres sont premiers ou non. On compte les non premiers et on renvoie une proportion laquelle ? sans oublier la liste des $faux$ actualisée.

Partie III. Compter les nombres premiers

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

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

Q15 On considère 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 $[n, \pi(n)]$.

Commencer, à la main, à calculer $Pi(4)$.

Écrire en PYTHON la fonction $Pi(N)$.
Un seul appel à *erato_iter* est autorisé et le paramètre N est un entier supérieur ou égal à 1.

Remarque : pour construire cette fonction, commencer par stocker la valeur de $erato_iter(N)$ dans une variable *table*. Puis, on construit également un tableau nb tel que, en fin de boucle, pour tout entier $i \geqslant 1$, la liste $[i, \pi(i)]$ est stockée dans la cellule d'indice $i-1$ (donc dans $nb[i-1]$). Le tableau nb intialement contiendra la liste des listes $[k, 0]$ pour k variant de 1 à $len(table)$. Dans la boucle, à chaque fois que $table[i]$ vaudra $True$, on ajoutera 1 à tous les $nb[j][1]$ pour j variant de i à $len(table)$. En effet, pour une position i donnée, pour chaque entier $j \geqslant i$ (on augmente d'une unité le compteur comptant les nombres premiers inférieurs ou égal à j. par exemple, si 3 est premier, il est comptabilisé dans les nombres premiers inférieurs ou égaux à 3 mais aussi à 4, à 5 etc.

Il a été prouvé que $\dfrac{n}{\ln(n)-1} < \pi(n)$ 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.

Remarque : Pour construire cette fonction PYTHON, on peut stocker au départ la liste produit par $Pi(N)$ dans une variable *test*. Il s'agit alors de comparer $test[n-1][1]$ et $n/(log(n)-1)$.

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 intégral li :

$$li : \mathbf{R}_+ \setminus \{1\} \to \mathbf{R},\ x \mapsto \int_0^x \frac{dt}{\ln(t)}.$$

Attention, si x franchit 1, donc si $x > 1$, l'intégrale généralisée $li(x)$ doit être interprétée comme sa valeur principale de Cauchy qui est définie comme :

$$li(x) = \lim_{\epsilon \to 0^+} \left(\int_0^{1-\epsilon} \frac{dt}{\ln(t)} + \int_{1+\epsilon}^{x} \frac{dt}{\ln(t)} \right) \quad (3)$$

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

$$\lim_{x \to +\infty} \frac{\pi(\lfloor x \rfloor)}{li(x)} = 1.$$

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

Q17 On choisit d'utiliser la méthode des rectangles à droite. Soit f une fonction continue sur $[a, b]$ avec $a < b$. Notons h le pas (largeur des rectangles). On découpe $[a, b]$ en une union d'intervalles $([a_k, a_{k+1}]_{0 \leqslant k \leqslant N}$ avec $a_0 = a$, $a_N = b$ et $a_{k+1} = a_k + h$. L'approximation de $\displaystyle\int_a^b f(t)\,dt$ est $\displaystyle\sum_{k=1}^{N} h\, f(a_k)$.

Quelle est la complexité de la fonction rectangles à droite en fonction de N ?

Q18 On considère maintenant la méthode des rectangles centrés.

L'approximation de $\displaystyle\int_a^b f(t)\,dt$ est $\displaystyle\sum_{k=0}^{N-1} h\, f\left(\frac{a_k + a_{k+1}}{2}\right)$.

Quelle est la complexité de la fonction rectangles centrés en fonction de N ?

Même question avec la méthode des trapèzes.

L'approximation de $\displaystyle\int_a^b f(t)\,dt$ est $\displaystyle\sum_{k=0}^{N-1} h\,\left(\frac{f(a_k) + f(a_{k+1})}{2}\right)$.

Q19 Écrire une fonction PYTHON $inv\_ln\_rect\_d(a, b, pas)$ qui calcule par la méthode des rectangles à droite une valeur approchée de $\displaystyle\int_a^b \frac{dt}{\ln(t)}$ avec un incrément valant pas. On suppose que $a < b$ et que 1 n'appartient pas à $[a, b]$. On considère que le réel $b - a$ est un multiple du réel pas.

Remarque : on peut utiliser $np.arange(a, b, pas)$ de $Numpy$.

Q20 Écrire une fonction $li\_d(x, pas)$ qui calcule une valeur approchée de $li(x)$ 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 $\epsilon = pas$ dans la formule (3) (valeur principale de Cauchy de $li(x)$). Les paramètres x et pas sont des flottants.

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 palier ce problème.

$$Ei : \mathbf{R}^\star \to \mathbf{R},\ x \mapsto \int_{-\infty}^{x} \frac{e^t}{t}\,dt.$$

Pour le cas $x > 0$, on utilise la valeur principale de Cauchy telle que vue pour li.

Le lien entre li et Ei est pour $x > 0$,

$$li(x) = Ei(\ln(x)).$$

Afin d'évaluer numériquement la valeur de Ei en un point, on se base sur son développement en série de Puiseux sur $]0, +\infty[$:

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

Avec $\gamma \equiv 0.5772156664901$ la constante d'Euler-Mascheroni.

Comme l'évaluation de la somme jusqu'à l'infini est impossible, on utilise en pratique la somme suivante :

$$Ei_n(x) = \gamma + \ln(x) + \sum_{k=1}^{n} \frac{x^k}{k \times k!}.$$

Le choix de n se fait en comparant Ei_{n-1} à Ei_n jusqu'à ce qu'ils soient considérés comme suffisamment 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 calculs et d'autres méthodes doivent être utilisées.

Q21 Écrire une fonction PYTHON $fact(n)$ qui renvoie $n!$ (on pourra faire une version itérative ou récursive au choix).

Écrire une fonction PYTHON $li\_dev(x)$ qui calcule $li(x)$ en se basant sur Ei_n, la fonction $sont\_proches$ de la question 2 et la fonction $fact$. La fonction li_dev doit renvoyer $False$ si :

- Ei_{n-1} et Ei_n ne peuvent être considérés comme proches au bout de $MAXIT = 100$ itérations.
- la valeur de x ne permet pas d'aboutir à un résultat.

Par ailleurs, 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 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
...		
ahubniggurath42	133	16
azathoth137	85	8

La seconde table est (fonctions et stocke de l'information 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 l'on teste une fonction de géneration 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

Q22 Expliquer pourquoi il n'est pas possible d'utiliser l'attribut *nom* comme clé primaire de la table **fonctions**.

Q23 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.