

Exercice 1

1)

```
def reindexation(f,xM,xN):  
    if f(xM)>f(xN):  
        (xM,xN)=(xN,xM)  
    return xM,xN
```

2) Par hypothèse, il est inutile de réindexer le segment [MN] dans cette fonction.

```
def iteration(f,xM,xN):  
    xR=2*xM-xN  
    if f(xR)<f(xM):  
        xN=xR  
    else:  
        xC1=(xM+xN)/2  
        xC2=(xM+xR)/2  
        if f(xC1)<f(xM):  
            xN=xC1  
        else:  
            xN=xC2  
    return xM,xN
```

3)

```
def ecart(xM,xN):  
    return 2*abs(xM-xN)/(abs(xM)+abs(xN))
```

4) Attention ici, après le while on écrit la négation de la condition d'arrêt

« l'écart est inférieur à eps ou bien le nombre d'itérations atteint Itmax »

ce qui donne

« l'écart est supérieur à eps et le nombre d'itérations est inférieur à Itmax »

avec des inégalités strictes.

```
def boucle(f,a,b,Itmax,eps):  
    xM,xN=(a,b)  
    nb_It=0  
    while nb_It<Itmax and ecart(xM,xN)>eps:  
        (xM,xN)=reindexation(f,xM,xN)  
        (xM,xN)=iteration(f,xM,xN)  
        nb_It=nb_It+1  
    return (xM,xN)
```

Exercice 2

Q1 – On commence par sélectionner les participants nés entre 1999 et 2003, puis on emploie une fonction d'agrégation pour les compter :

```
SELECT COUNT(*) FROM Participant
      WHERE 1999 <= ne AND ne <= 2003
```

On peut aussi utiliser la clef primaire pid pour compter les participants :

```
SELECT COUNT(pid) FROM Participant
      WHERE 1999 <= ne AND ne <= 2003
```

Q2 – On regroupe les randonnées par niveau de difficulté, puis on applique une fonction d'agrégation à chacun des groupes obtenus :

```
SELECT diff, AVG(duree) AS d_moy FROM Rando GROUP BY diff
```

Q3 – On sélectionne les participants dont le niveau de difficulté maximum est inférieur à celui de la randonnée n°42, en récupérant ce dernier grâce à une sous-requête :

```
SELECT pnom FROM Participant
      WHERE diff_max < (SELECT diff FROM Rando WHERE rid == 42)
```

On peut aussi utiliser le produit cartésien des deux tables puis effectuer une sélection portant sur les niveaux de difficulté :

```
SELECT pnom FROM Participant, Rando
      WHERE rid == 42 AND diff_max < diff
```

Q4 – Première possibilité : on joint la table Rando avec elle-même (en identifiant les noms des randonnées) et on détecte les homonymies grâce à la présence de deux clefs distinctes pour une même nom :

```
SELECT DISTINCT R.rid FROM Rando R JOIN Rando S
      ON R.rnom == S.rnom
      WHERE R.rid != S.rid
```

Seconde possibilité : on utilise une sous-requête qui renvoie, grâce à une agrégation, la liste des noms apparaissant plusieurs fois (on regroupe les randonnées par nom, et on dénombre les paquets) :

```
SELECT DISTINCT rid FROM Rando WHERE rnom IN
      (SELECT rnom FROM rando
      GROUP BY rnom HAVING COUNT(*) > 1)
```

Exercice 3

Q1. Il suffit d'importer directement les fonctions désirées de la façon suivante :

```
from math import log, sqrt, floor, ceil
print(log(0.5))
```

On aurait également pu importer la bibliothèque math et procéder comme ceci :

```
import math as m
print(m.log(0.5))
```

Q2) On peut renvoyer directement la condition logique, qui a une valeur booléenne :

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

Q3) Comme $1001 > 10$, `mystere(1001, 10)` renvoie $1 + \text{mystere}(100.1, 10)$.

Comme $100.1 > 10$, `mystere(100.1, 10)` renvoie $1 + \text{mystere}(10.01, 10)$.

Comme $10.01 > 10$, `mystere(10.01, 10)` renvoie $1 + \text{mystere}(1.001, 10)$.

Enfin $1.001 < 10$ donc `mystere(1.001, 10)` renvoie 0

Ainsi,

`mystere(1001, 10)` renvoie 3.

Q4) Elle permet de voir combien de fois il faut diviser x par b pour qu'il soit strictement plus petit que b . Autrement dit, elle renvoie l'unique entier naturel n tel que $b^n \leq x < b^{n+1}$.

Alors
$$n \ln(b) \leq \ln(x) < (n+1) \ln(b) \quad \text{par stricte croissance de } \ln$$

d'où
$$n \leq \frac{\ln(x)}{\ln(b)} < n+1 \quad \text{puisque } \ln(b) > 0$$

si bien que n est la partie entière de $\frac{\ln(x)}{\ln(b)} = \log_b(x)$, le logarithme de x en base b .

De ce fait,

`mystere(x, b)` renvoie $\lfloor \log_b(x) \rfloor$.

Comme la fonction log admet la base comme argument optionnel, il suffit de taper

```
def mystere2(x,b):
    return floor(log(x,b))
```

Q5) Lors de la dernière étape de la boucle, la variable i prend la valeur 99 999 donc x_1 vaut $100\,000 \times \text{pas} \dots$ soit 1. C'est bien la valeur affichée en sortie de boucle.

La variable x_2 est initialisée à 0 puis on lui ajoute 100 000 fois la variable *pas*. Le codage binaire de 10^{-5} n'étant qu'une valeur approchée, cette succession de 100 000 additions va accumuler des erreurs de calculs sur les dernières décimales ; ceci explique que x_2 ne soit pas exactement égal à 1.

Q6) Un octet étant un paquet de 8 bits, 32 bits occupent 4 octets. Avec 4 Go de mémoire vive, on pourra donc stocker plus d'un milliard de booléens. Ainsi, la valeur maximale approximative de N est 10^9 .

Q7) Les booléens ne prenant que deux valeurs possibles, on peut les coder sur un seul bit (avec 0 pour False et 1 pour True par exemple). Cela permet de gagner un facteur 32 sur la valeur maximale de N.

Q8) On traduit le pseudo-code en Python. On suppose que les diverses fonctions utiles ont déjà été importées (cf. question 1) et, pour la partie entière, on peut indifféremment utiliser `int` ou `floor`, qui auront le même effet sur un réel positif :

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

Q9) Dans ce cas, on ajoute 2^i à chaque itération, si bien que la valeur finale de A est

$$A = \sum_{i=0}^{N-1} 2^i = 1 \times \frac{2^N - 1}{2 - 1} = \boxed{2^N - 1}.$$

C'est la valeur maximale renvoyée par l'algorithme pour une valeur donnée de N.

Q10) Il faut penser à d'abord importer la fonction `time()`. Jouer ensuite avec `int()` pour récupérer la partie fractionnaire, que l'on multiplie par 10^7 puis arrondit.

```
from time import time
def IZ(N):
    p1 , p2 = 24375763 , 28972763
    M = p1 * p2
    t = time()
    x = int( 10**7 * (t-int(t)) )
    A = 0
    for i in range(N):
        if x%2 == 1:
            A = A + 2**i
        x = x**2 % M
    return A
```

Q11) On utilise les idées exposées en début de partie : il suffit de réitérer l'algorithme IZ jusqu'à ce que le nombre obtenu passe le test de Fermat pour les valeurs $a \in \{2; 3; 5; 7\}$. Ce dernier point implique au passage que ce nombre soit au moins égal à 8.

Attention au choix de N : on a vu dans la question 9 que la valeur maximale du nombre A renvoyé par IZ(N) est $2^N - 1$. Il faut donc que N vérifie

$$2^N - 1 < n_{\max} \iff 2^N < n_{\max} + 1 \iff N < \frac{\ln(n_{\max} + 1)}{\ln(2)} = \log_2(n_{\max} + 1)$$

On prendra ainsi $N = \lfloor \log_2(n_{\max} + 1) \rfloor - 1$.

```
def premier_rapide(n_max):
    N=int(log(n_max+1,2)) - 1
    while True:
        # Tirage d'un entier pseudo-aléatoire p
        p = IZ(N)
        if p > 7:
            # Test de primalité
            premier = True
            for a in (2,3,5,7):
                premier = premier AND (a**(p-1) % p == 1)
            if premier:
                # On interrompt la fonction en renvoyant p
                return p
```

Q12) On fait évidemment appel au crible d’Eratosthène pour vérifier que les nombres manipulés sont premiers. Pour rappel : celui-ci renvoie une liste de booléens `list_bool` et les nombres premiers sont les indices des termes de `list_bool` égaux à `True`.

```
def stats(N,nb):
    premiers = eratosthene(N)
    erreurs = 0
    non_premiers = []
    for i in range(nb):
        p = premier_rapide(N)
        if not premiers[p]:
            erreurs = erreurs+1
            non_premiers.append(p)
    return erreurs/nb , non_premiers
```

Q13) Pour ne pas faire exploser la complexité, il ne faut effectuer qu’un seul appel à la fonction `eratosthene()`, puis une seule boucle qui parcourt les entiers de 0 à N.

```
def pi(N):
    premiers = eratosthene(N)
    pi = 0
    L = []
    for n in range(N+1):
        if premiers[n]:
            pi = pi+1
            L.append((n,pi))
    return L
```

Q14) On va créer d’abord la liste des valeurs de $\pi(n)$ par compréhension.

```
def verif_pi(N):
    l_pi = [e[1] for e in pi(N)]
    n = 5393
    res = True
    while res and n <= N:
        res = l_pi[n] > n/(log(n)-1)
        n = n+1
    return res
```