BANQUE PT 2022 : CORRIGE DE LA PARTIE INFORMATIQUE DE L'EPREUVE D'INFORMATIQUE ET MODELISATION DE SYSTEMES PHYSIQUES

SUIVI DE LA CONSOMMATION D'EAU D'UNE HABITATION

SECONDE PARTIE : EXPLOITATION INFORMATIQUE DES MESURES

C) ESTIMATION DES DEBITS ET VOLUMES (30 % du barème de l'épreuve)

1) Justification du format des données

Q22. On a $Q = \frac{c}{\tau}$, donc $\tau = \frac{c}{Q}$, donc $\tau_{min} = \frac{c}{Q_{max}} = \frac{7.10^{-5}}{2,5} = 3.10^{-5}\ h = 3.10^{-5} \times 3600 = 0,1\ s$

$$\boxed{\tau_{min} = \frac{c}{Q_{max}} = 0,1\ s}$$

Q23. On a $Q = \frac{c}{\tau}$, donc $\frac{\delta Q}{Q} = \sqrt{\left(\frac{\delta c}{c}\right)^2 + \left(\frac{\delta \tau}{\tau}\right)^2}$.

En supposant qu'il n'y a pas d'incertitude sur le volume d'eau *c*, alors $\boxed{\frac{\delta Q}{Q} = \frac{\delta \tau}{\tau}}$

On veut $\frac{\delta Q}{Q} < 1\%$, donc $\frac{\delta \tau}{\tau} < 1\%$

Pour le débit Q_{max}, $\tau = \tau_{min}$, il faut donc $\delta\tau < 1\% . \tau_{min} = \frac{0,1}{100} = 0,001\ s = 1\ ms$

$$\boxed{\delta\tau_{max} = 1\ ms}$$

Q24.

Avec le stockage entier :

Les instants exprimés en ms sont de la forme $1620115201145 > 10^{12} = (10^3)^4 \approx (2^{10})^4 = 2^{40} > (2^{31} - 1)$.
Il y a donc un overflow. Il n'est pas possible de stocker directement les instants des impulsions dans le microcontrôleur avec la résolution demandée, avec le stockage entier.

Avec le stockage flottant :

$1620115201145 > 10^{12} = (10^3)^4 \approx (2^{10})^4 = 2^{40}$.
Il faudrait donc une mantisse codée sur plus de 40 bits pour avoir la résolution demandée, or ici la mantisse n'est codée que sur 23 bits.
Il n'est pas possible de stocker directement les instants des impulsions dans le microcontrôleur avec la résolution demandée, avec le stockage flottant.

2) Récupération des données

Q25.

```
def recup(nomFichier):
    # on commence par récupérer les données comme le propose l'énoncé
    f = open(nomFichier, "r")
    lignes = f.readlines()
    f.close()

    lImp = []  # liste qui contiendra les timestamp
    t_init = float(lignes[0])   # on convertit en flottant
    # on ajoute ensuite les autres instants :
    for k in range(1, len(lignes)):
        lImp.append(t_init + float(lignes[k])/1000)
    return lImp
```

3) Première reconstruction : débit constant par morceaux

Q26.

```
def debits1(lImp, c):
    lQ = []  # liste qui contiendra les débits
    for i in range(len(lImp)-1):
        lQ.append(c / (lImp[i+1] - lImp[i]))
    return lQ
```

Q27. *La relation (3) propose :* $q(t)=\begin{cases}0 & \text{si } t<T_0 \quad ou \quad t\geq T_{n-1}\\ q_i & \text{si } T_i\leq t<T_{i+1}\ ,\ 0\leq i<n-1\end{cases}$

Il me semble qu'il aurait été plus judicieux de proposer : $q(t)=\begin{cases}0 & \text{si } t<T_0 \quad ou \quad t\geq T_n\\ q_i & \text{si } T_i\leq t<T_{i+1}\ ,\ 0\leq i<n\end{cases}$

Quoiqu'il en soit, on peut proposer le code suivant pour la fonction indice :

```
def indice(lImp, t):
    i = 0
    while t >= lImp[i+1]:
        i = i + 1
    return i
```

Q28.

```
def temporel(lQ, lImp, lTemps):
    lQT = []  # liste qui contiendra les débits moyens
    n = len(lImp)-1
    for k in range(len(lTemps)):
        if lTemps[k] < lImp[0] or lTemps[k] >= lImp[n-1]:
            lQT.append(0)
        else:
            lQT.append(lQ[indice(lImp, lTemps[k])])
    return lQT
```

Autre possibilité, en parcourant la liste par compréhension :

```
def temporel2(lQ, lImp, lTemps):
    lQT = []  # liste qui contiendra les débits moyens
    n = len(lImp)-1
    for t in lTemps:
        if t < lImp[0] or t >= lImp[n-1]:
            lQT.append(0)
        else:
            lQT.append(lQ[indice(lImp, t)])
    return lQT
```

Q29. Dans la fonction *temporel*, on parcourt *m* fois la boucle *for*.
Dans le pire des cas, tous les instants *t* de la liste *lTemps* sont compris entre *lImp[n-2]* et *lImp[n-1]*. L'appel à la fonction indice est alors de complexité *n*.
Dans le pire des cas, la complexité asymptotique de la fonction *temporel* est donc en $O(m \times n)$.

Q30. A l'étape *k*, l'instant *t = lTemps[k]* est tel que *lImp[i] < t < lImp[i+1]*
A l'étape *k*+1, en appelant la fonction indice, on compare à chaque fois le nouvel instant *t = lTemps[k+1]* à *lImp[1], lImp[2], lImp[3], ..., lImp[i],* alors que c'est inutile. On peut commencer à comparer à *lImp[i+1]*.
Dans ce cas, la complexité est en $O(n)$.
On passe donc d'une complexité quadratique dans le pire des cas à une complexité linéaire.

Q31. On a $v_j - v_{j-1} = \int_{v_{j-1}}^{v_j} dv = \int_{t_{j-1}}^{t_j} q\,dt$

Par la méthode des trapèzes : $v_j - v_{j-1} = \frac{q_{j-1} + q_j}{2} \times (t_j - t_{j-1})$

$$v_j = v_{j-1} + \frac{q_{j-1} + q_j}{2} \times \Delta t$$

Q32.

```
def volumes(lTemps, lQT, v0):
    lVT = [v0]  # liste qui contiendra les volumes consommés
    v = v0
    # on calcule deltat une bonne fois pour toute, les instants de calcul
    # étant supposés régulièrement espacés :
    deltat = lTemps[1] - lTemps[0]
    for j in range(1, len(lTemps)):
        v = v + (lQT[j] + lQT[j-1]) / 2 * deltat
        lVT.append(v)
    return lVT
```

Q33.

```
lImp = recup("2021-01-31.txt")
lQ = debits1(lImp, 0.07)
lTemps = np.linspace((lImp[0], lImp[-1]), 1000)  # tableau Numpy
lQT = temporel(lQ, lImp, lTemps)
lVT = volumes(lTemps, lQT, 0)
```

Q34. *Les graphes de la figure 4 ne sont pas cohérents. En effet par exemple, le rectangle hachuré (dans le graphe en bas à droite de la figure 4) devrait être de même largeur que le rectangle non hachuré, c'est-à-dire :* $(T_{i\ nouveau} - T_{i-1}) = 2\ (T_{i\ ancien} - T_{i-1})$, *soit* $T_{i\ nouveau} = 2\ T_{i\ ancien} - T_{i-1}$. *En effet, le volume consommé entre deux impulsions est toujours le même et égal à c.*
Par ailleurs, l'énoncé passe d'une notation lImp à Limp… Changement de notation sans intérêt et qui peut apporter de la confusion…

```
def debits2(Limp, c, seuil):
    Limp_modif = Limp[:]
    lQ = []  # liste qui contiendra les débits
    modif = False
    for i in range(len(Limp)-1):
        qi = c / (Limp[i+1] - Limp[i])
        if qi > seuil or modif:
            lQ.append(qi)
            modif = (qi <= seuil)   # on change éventuellement modif
        else:
            lQ.append(0)
            # on modifie Ti de sorte à conserver le volume total consommé
            Limp_modif[i] = 2 * Limp[i] - Limp[i-1]
            modif = True    # on change modif
    return (Limp_modif, lQ)
```

D) TRANSMISSION DES DONNEES ET CONSULTATION A DISTANCE (30 % du barème de l'épreuve)

1) CHIFFREMENT ET DECHIFFREMENT DES DONNEES

Q35.

$13 = 8 + 4 + 1 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (1101)_2$

$7 = 4 + 2 + 1 = 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = (0111)_2$

Le OU exclusif bit à bit donne alors 1^0 (poids fort) puis 1^1 puis 0^1 puis 1^1 (poids faible), soit $(1010)_2$, soit $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 8 + 2 = 10$.

L'instruction 13^7 renvoie 10.

Q36.

```
def code1(L, cle):
    Lc = []  # liste qui contiendra les éléments de L chiffrés à l'aide de cle
    for i in range(len(L)):
        Lc.append(L[i] ^ cle)
    return Lc
```

Q37.

	a = 0	a = 1
b = 0	a ⊕ b = 0 (a ⊕ b) ⊕ b = 0	a ⊕ b = 1 (a ⊕ b) ⊕ b = 1
b = 1	a ⊕ b = 1 (a ⊕ b) ⊕ b = 0	a ⊕ b = 0 (a ⊕ b) ⊕ b = 1

On constate dans tous les cas que (a ⊕ b) ⊕ b = a.

On note : *a* un bit du message, *b* le bit de la clé correspondant, et *c* le bit du message chiffré correspondant.

On a donc $c = (a \oplus b)$.

On sait que : $(a \oplus b) \oplus b = a$.

On obtient donc $a = c \oplus b$.

Le bit du message est obtenu en appliquant un OU exclusif entre le bit du message chiffré correspondant et le bit de la clé correspondant.

On peut ainsi reconstruire le message, en appliquant cette méthode pour tous les bits, bit à bit.

Il suffit d'appeler *code1(Lc, cle)* pour obtenir L.

Cet algorithme de chiffrement est dit *symétrique* car si *Lc = code1(L, cle)*, alors *L = code1(Lc, cle)*.

La fonction utilisée pour décoder est la même que pour coder, avec la même clé.

Q38.

```
def bin64(n):
    chaine_0b = bin(n)
    # on enlève les 2 permiers caractères (0b)
    chaine = chaine_0b[2:]
    # on ajoute des 0 devant par concaténation :
    for i in range(64-len(chaine)):
        chaine = "0" + chaine
    return chaine
```

Autre possibilité :

```
def bin64_bis(n):
    chaine_0b = bin(n)
    # on enlève les 2 permiers caractères (0b)
    chaine = chaine_0b[2:]
    # on ajoute des 0 devant et on ajoute chaine (par concaténation) :
    return "0" * (64 - len(chaine)) + chaine
```

Q39.

```
def bits(L):
    chaine = ''
    for i in range(len(L)):
        chaine = chaine + bin64(L[i])
    return chaine
```

Autre possibilité, en parcourant la liste par compréhension :

```
def bits2(L):
    chaine = ''
    for c in L:
        chaine = chaine + bin64(c)
    return chaine
```

Q40.

```
def entiers(flux):
    L = []  # liste qui contiendra les entiers codés
    nombre_d_entiers = len(flux) // 64  # nombre d'entiers codés
    for i in range(nombre_d_entiers):
        L.append(int(flux[64*i: 64*(i+1)], 2))
    return L
```

Q41. La position du bit de la clé correspondant au bit de rang i de la liste à chiffrer est $i \% l$ (reste de la division euclidienne de i par l, l étant la longueur de la clé).

Q42.

```
def code2(L, cle):
    cle_binaire = bin(cle)  # on commence par écrire la clé en binaire
    cle_bin = cle_binaire[2:]   # on enlève les 2 permiers caractères (0b)
    if len(cle_bin) % 2 == 0:   # si cle_bin est de longueur paire
        cle_bin = "0" + cle_bin     # on ajoute un 0 par concaténation
    l = len(cle_bin)
    chaine_L = bits(L)  # chaine contenant les informations de L en binaire
    chaine_cod = ''   # contiendra le résultat du OU exclusif bit à bit
    for i in range(len(chaine_L)):
        chaine_cod = chaine_cod + str(int(chaine_L[i]) ^ int(cle_bin[i % l]))
    return entiers(chaine_cod)
```

Q43.

Chiffrement par la fonction *code 1* :

On note : a un bit du message, b le bit de la clé correspondant, et c le bit du message chiffré correspondant.

On a donc $c = (a \oplus b)$.

On sait que : $(a \oplus b) \oplus b = a$, donc $c \oplus b = a$.

On applique un OU exclusif à cette relation : $(c \oplus b) \oplus c = a \oplus c$.

Or $(c \oplus b) \oplus c = (b \oplus c) \oplus c = b$.

On obtient $b = a \oplus c$.

Le bit de la clé correspondant est obtenu en appliquant un OU exclusif entre le bit du message et le bit du message chiffré correspondant.

On peut ainsi reconstruire la clé, bit par bit.

Chiffrement par la fonction *code 2* :

De la même manière, on peut reconstituer la répétition de la clé (exprimée en binaire). Il reste alors simplement à chercher la longueur de la clé (exprimée en binaire) : on commence par faire l'hypothèse d'une clé de longueur 1, puis 3, puis 5, … (si on a utilisé la même méthode, la clé est de longueur impaire), en regardant à chaque fois si la chaine ainsi créée coïncide avec la reconstitution de la répétition de la clé obtenue par la méthode décrite précédemment.

2) CONSULTATION DES DONNEES

Q44.

1. SELECT conso
   FROM eau
   WHERE date = '2017-11-17'

2. SELECT date
   FROM eau
   WHERE conso > 500

Q45.

1. SELECT conso
   FROM eau
   WHERE date LIKE '2017-11-%'

2. SELECT SUM(conso), AVG(conso)
   FROM eau
   WHERE date LIKE '2017-11-%'

   En utilisant des alias pour plus de clarté :

   SELECT SUM(conso) AS totale, AVG(conso) AS moyenne
   FROM eau
   WHERE date LIKE '2017-11-%'

Q46. *Cette question n'est pas très claire…*

Pour la table *habitations*, il faudra les attributs suivants :

- Un identifiant *id* associé au logement ;
- Un attribut *nom* associé à la personne logée (et éventuellement d'autres attributs donnant des renseignements sur la personne logée ou sur le logement, comme *prenom*, *date_de_naissance*, *adresse* etc).

Dans la table *eau*, il faut ajouter un identifiant *id_compteur* associé à chaque compteur.

Pour effectuer des requêtes sur ces deux tables, il faudra utiliser une condition de jointure :

SELECT …
FROM eau JOIN habitations ON eau.id_compteur = habitations.id
WHERE …