Correction

Exercice 1 SQL (E3A MP 2018).

Correction :

1° La table `zoos` possède une clef primaire qui est `id` et pas de clef étrangère. La table `animaux` possède une clé primaire qui est `id` et une clef étrangère qui est `zoo`.

```
2° SELECT id, nom, naissance, zoo
   FROM animaux
   WHERE naissance < 2010
```

```
3° SELECT id, nom, naissance, zoo
   FROM animaux
   WHERE espece = "Chameau" AND sexe = "F"
   ORDER BY nom ASC
```

```
4° SELECT animaux.nom AS nom_animal, animaux.espece, zoos.nom AS nom_zoo, zoos.pays
   FROM animaux JOIN zoos ON animaux.zoo = zoos.id
```

```
5° SELECT animaux.id, animaux.nom, animaux.naissance, animaux.zoo
   FROM animaux JOIN zoos ON animaux.zoo = zoos.id
   WHERE animaux.espece = "Bonobo" AND animaux.sexe = "M" AND zoos.continent = "Asie"
```

```
6° SELECT pays, COUNT(*)
   FROM zoos
   GROUP BY pays
```

```
7° SELECT pays
   FROM (SELECT DISTINCT pays, continent
         FROM zoos )
   GROUP BY pays
   HAVING COUNT(*) > 1
```

Exercice 2 Code correcteur (Banque PT 2015).

Correction :

1° Q1. Comme (on souligne l'écriture en binaire) $5 = \underline{101}$, son bit de parité est 0, comme $16 = \underline{10000}$ son bit de parité est 1 et comme $37 = \underline{100101}$ son bit de parité est aussi 1.

```
Q2. def parite(bits):
    return sum(bits) % 2
```

Q3. Si deux bits (plus généralement un nombre pair de bits) sont altérés alors l'erreur n'est pas détectable. Si un seul bit a été altéré il n'y a pas moyen de détecter où se trouve le bit altéré.

```
2° Q4. def encode_hamming(donnee):
    d1, d2, d3, d4 = donnee # pour simplifier l'écriture
    p1 = parite([d1, d2, d4])
    p2 = parite([d1, d3, d4])
    p3 = parite([d2, d3, d4])
    return [p1, p2, d1, p3, d2, d3, d4]
```

```

Q5. def decode_hamming(message):
    m1, m2, m3, m4, m5, m6, m7 = message # idem
    c1 = parite([m4, m5, m6, m7])
    c2 = parite([m2, m3, m6, m7])
    c3 = parite([m1, m3, m5, m7])
    if [c1, c2, c3] != [0, 0, 0]:
        numbit = 4*c1 + 2*c2 + c3
        print("Attention il y a une erreur dans la transmission")
        print("Correction du bit numéro :", numbit)
        m = message.copy()
        m[numbit-1] = 1 - m[numbit-1]
        m1, m2, m3, m4, m5, m6, m7 = m # pour éviter de mettre des [] partout
    return [m3, m5, m6, m7]

```

Q6. La donnée 1001 se code en 0110011, le "les deux premiers bits" de l'énoncé n'est pas très claire, s'il s'agit des poids faibles le message reçu est 0110000 dans ce cas, il y a correction du bit numéro 1 et on a 1000, et s'il s'agit des poids forts le message reçu est 1010011 et dans ce cas, il a correction du bit numéro 3 et on a 0011.

Dans ce cas l'effet de la correction est mauvais

Q7. On rajoute un huitième bit de parité sur l'ensemble des 7 bits. On suppose qu'il n'y a jamais plus de deux erreurs. Ainsi si $c1=c2=c3=0$ alors c'est bon et on décode, sinon si le 8e bit est mauvais alors il n'y a qu'une erreur sur les sept premiers et on peut corriger, si le 8e bit est bon c'est qu'il y a une double erreur et on ne peut pas corriger.

Remarque : L'idée sous-jacente provient du fait qu'on peut voir un message de n bits comme un vecteur du $\mathbb{F}_2$ -espace vectoriel $\mathbb{F}_2^n$ (où $\mathbb{F}_2$ est le corps à 2 éléments, comme 2 est un nombre premier, $\mathbb{F}_2$ n'est rien d'autre que les entiers modulo 2). Pour pouvoir détecter et corriger des erreurs on va transmettre non pas nos n bits mais m bits (avec $m > n$). Pour cela on identifie l'ensemble des messages possibles à un sous-espace vectoriel F de dimension n du $\mathbb{F}_2$ -ev $\mathbb{F}_2^m$ (qui est de dimension m). On transmet notre message à m bits, si le message reçu est encore un élément de F on considère qu'il est bon (ce qui n'est pas nécessairement le cas... mais bon on choisit le code correcteur en fonction de la fiabilité de la transmission), sinon on cherche le mot de F le plus proche du mot reçu, pour cela on utilise la distance de Hamming (la distance de Hamming est le nombre de coordonnées différentes). L'idée est de choisir m et F de tel manière à maximiser la distance de Hamming minimale entre deux éléments de F qu'on note δ (on peut aussi maximiser l'entier r tel que les boules fermées de rayon r et centrées en les éléments de F soient disjointes, en fait r n'est rien d'autre que le plus grand entier strictement plus petit que $\frac{\delta}{2}$). Ces trois quantités n , m et δ sont reliées. Une référence possible sur le côté mathématique est le livre de Michel Demazure (Cours d'algèbre : primalité, divisibilité, codes).

Exercice 3.

Correction :

```

1° def mystere(i,j):
    t = 0
    while i>=1:
        if i%2!=0:
            t=t+j
            j = 2*j
            i = floor(i/2)
    return t

```

2°

```

>>> mystere(8,9)                >>> mystere(9,8)
i= 4 ,   j= 18, t= 0             i= 4 ,   j= 16, t= 8
i= 2 ,   j= 36, t= 0             i= 2 ,   j= 32, t= 8
i= 1 ,   j= 72, t= 0             i= 1 ,   j= 64, t= 8
i= 0 ,   j= 144, t= 72           i= 0 ,   j= 128, t= 72
72                                72

```

3° La valeur de i est toujours entière et décroît strictement, comme une suite strictement décroissante d'entier est toujours stationnaire à 0 on est garanti que i prendra la valeur 0 en un nombre fini d'étape.

4° Montrons que $i_n j_n + t_n$ à toujours la même valeur après chaque passage dans la boucle.

Si i_n est pair alors $i_{n+1} j_{n+1} + t_{n+1} = \frac{i_n}{2} 2j_n + t_n = i_n j_n + t_n$ et si i_n est impaire alors $i_{n+1} j_{n+1} + t_{n+1} = \frac{i_n-1}{2} 2j_n + t_n + j = i_n j_n + t_n$.

Ainsi la quantité $i_n j_n + t_n$ est un invariant de boucle. En sortie de boucle i contient 0, et donc t contiendra la valeur du produit ij (car en début de boucle t contient 0).

Remarque : cet algorithme permet de multiplier deux entiers en utilisant que des additions et des multiplications/divisions par 2 (un ordinateur travail en binaire ainsi multiplier par 2 c'est comme multiplier par 10 quand on travail en base 10). C'est ce type d'algorithme qui est utilisé par le processeur pour faire des multiplications.

Exercice 4.

Correction : La liste est croissante si pour tout k on a $t[k] \leq t[k+1]$, donc elle n'est pas croissante s'il existe un k tel que $t[k] > t[k+1]$, c'est ce qu'on fait

Code Python :

```
def est_croissante(t):
    res=True
    n=len(t)
    for k in range(n-1):
        if t[k] > t[k+1]:
            res=False
    return res
```