DS 3, Codes correcteurs d'erreurs : solutions

Q1) Le nombre de messages est le cardinal de $\{0,1\}^k$ donc 2^k.

Q2)
```
def parite(m):
    s=0
    for x in m:
        s=s+x
    return m+[s%2]
```

Q3)
```
def paritebloc(m):
    r=[]
    p=len(m)//8
    for i in range(p):
        s=0
        for j in range(8):
            r.append(m[8*i+j])
            s=s+m[8*i+j]
        r.append(s%2)
    return r
```

Ou encore, à l'aide de la fonction `parite` précédente :

```
def paritebloc2(m):
    l=len(m)
    n=l//8
    m1=[]
    for i in range(n):
        m1=m1+parite(m[8*i:8*(i+1)])
    return m1
```

Ou encore, de manière très concise en récursif (Baptiste) :

```
def paritebloc8(m):
    if m==[]:
        return m
    else :
        return parite(m[:8])+paritebloc8(m[8:])
```

Q4) Si un seul bit du bloc de neuf bits est modifié, le bit ne vérification ne sera plus égal à la somme modulo 2 des 8 premiers puisqu'une erreur sur l'un des huit premier bit change de 1 la somme modulo 2. (Si l'erreur est sur le bit de vérification, il n'est bien sûr plus égal à la somme non plus).

En revanche, on ne sait pas sur quel bit l'erreur s'est produite. On dit que ce codage *détecte* une erreur, mais il n'est pas capable de *corriger* une erreur.

Q5) **a)** Pour 000 :

- Avec une erreur : on peut recevoir 001, 010, 100.
- Avec deux erreurs : 011, 101, 110.
- Avec trois erreurs : 111.

Pour 111 :

- Avec une erreur : 110, 101, 011.

— Avec deux erreurs ; 100, 010, 001

— Avec trois erreurs 000

b) Dans la liste précédente, un message codé étant donné, la probabilité de chaque message avec une erreur est $p(1-p)^2$ donc la probabilité d'avoir une erreur est $3p(1-p)^2$. En revanche la probabilité de chaque message avec deux erreurs est $(1-p)p^2$ donc la probabilité d'avoir deux erreurs est $3(1-p)^2p$.
Comme $p < 1/2$, on sait que $(1-p) < p$ donc que

$$3(1-p)p(1-p) = a.(1-p) < 3(1-p)p^2 = a.p,$$

donc la probabilité d'avoir une erreur est supérieure à celle d'avoir deux erreurs.

c) Si on reçoit 101 on sait qu'au moins une erreur s'est produite, et comme on vient de dire qu'il est plus probable qu'il n'y en ait une qu'une seule, on conclut que le message codé le plus probable était 111.

d) Si trois erreurs (une sur chaque bit) se sont produites, on reçoit 111 ou 000 donc on message qui est un message du code, donc on ne peut pas détecter que des erreurs se sont produites.

Q6) On peut bien sûr parcourir les indices de m_1 et, dans le cas $m_1[i] \neq m_2[i]$, incrémenter la distance de 1. On peut aussi simplement rajouter la différence $|m_1[i] - m_2[i]|$ qui fait 0 ou 1.

```
def dist(m1,m2):
    s=0
    for i in range(len(m1)):
        s=s+abs(m1[i]-m2[i])
    return s
def pds(m):
    zero=[0]*len(m)
    return dist(zero,m)
```

La seconde fonction tient compte du fait que le poids d'un message est sa distance avec le code $(0, \ldots, 0)$.

Q7)

```
def distmini(m,C):
    mini=dist(m,C[0])
    L=[C[0]]
    for i in range(1,len(C)):
        r=dist(m,C[i])
        if r<mini:
            L=[C[r]]
            mini=r
        elif r==mini:
            L.append(C[r])
    return mini
```

Autre méthode, avec deux boucles, mais sans avoir besoin de réaffecter plusieurs fois des listes, donc pas forcément moins bien !

```
def distmini2(m,C):
    d=dist(m,C[0])
    L=[]
    for i in range(1,len(C)):
        dc=dist(m,C[i])
        if dc<d:
            d=dc
    for i in range(len(C)):
        dc=dist(m,C[i])
```

```
        if dc==d:
            L.append(C[i])
    return d,L
```

Q8) Comme remarqué à la Q6, pour deux mots m_1 et m_2 de longueur k,

$$d(m_1, m_2) = \sum_{i=1}^{k} |m_1[i] - m_2[i]|.$$

On peut donc voir d comme restriction à $\{0,1\}^k$ de la distance définie par la norme 1 sur $\mathbb{R}^k$ et l'inégalité triangulaire est donc connue pour cette norme.
Refaisons néanmoins la preuve, déduite de l'I.T. dans $\mathbb{R}$: soit m_1, m_2, m_3 trois mots de longueur k,

$$\begin{aligned} d(m_1, m_3) &= \sum_{i=1}^{k} |m_1[i] - m_3[i]| \\ &= \sum_{i=1}^{k} |m_1[i] - m_2[i] + m_2[i] - m_3[i]| \\ &\leq \sum_{i=1}^{k} |m_1[i] - m_2[i]| + |m_2[i] - m_3[i]| \\ &= d(m_1, m_2) + d(m_2, m_3) \end{aligned}$$

Q9) On peut bien sûr utiliser la fonction précédente en ne gardant que la second valeur de retour :

```
C=[[0]*7,[1]*7]
def deconaif(m):
    return distmini(m,C)[1]
```

Une autre méthode ici : Il suffit ici de compter le nombre de 1 dans le message codé : s'il est supérieur ou égal à 4, on considère que le message original est 1, sinon c'est 0.
L'intérêt de la longueur impaire 7 pour les messages codés vient du fait qu'il n'y a pas de message qui serait à égale distance (de Hamming) de 0000000 et 1111111.

Q10) Pour un codage de type (n, k), le code contient 2^k messages.
Sachant que nous devons tester la distance avec tous les 2^k messages du code et que nous devons effectuer $O(n)$ opérations pour déterminer une distance entre deux messages de longueur n . Nous obtenons une complexité en $O(n2^k)$.

Q11) Comme nous venons de le déterminer à la question précédente, un coût d'un tel algorithme est exponentiel en la dimension du message, ce qui n'est pas raisonnable. De plus, si plusieurs messages peuvent réaliser ce minimum, le message obtenu par cette méthode n'a aucune raison d'être le message d'origine. Donc pour pouvoir *corriger* un message reçu, il doit n'y avoir qu'un message du code à distance minimale du message reçu.

Q12) Nous obtenons en considérant les messages 000000000 et 000000011 que la distance est inférieure à 2. Il suffit de montrer que deux messages m et m' distincts du code ont une distance supérieure à 2. Or deux messages distincts du code proviennent de deux messages d'origine distincts. La distance entre les deux messages d'origine est donc d'au moins 1. Si cette distance est égale au moins à 2 la distance entre m et m' est nécessairement (codage systématique) au moins égale à 2. Si cette distance est égale à 1 les mots diffèrent d'un bit et par suite leur bit de parité est nécessairement distinct. Donc la distance entre m et m' est égale à 2. Dans tous les cas nous obtenons que la distance est supérieure à 2.

Q13) Si le nombre d'erreur est inférieur ou égal à $d-1$, la distance entre le message reçu et le code est inférieur ou égale à $d-1$, et donc ce n'est pas un message du code. On sait donc que le message reçu est erroné.

En revanche dès que le nombre d'erreur dépasse d, il se peut qu'à partir du message du code les erreurs aient fabriqué un autre message qui est dans le code, on ne peut donc ne pas voir qu'il s'agit d'un message erroné.

Q14) Comme dit à la fin de la Q10, pour corriger un message m erroné, on doit être sûr qu'il n'y a qu'un message du code à distance minimale de m.

Or si m est un message contenant p erreurs avec $2p < d \quad (*)$ alors si m_1 et m_2 sont deux mots du code à distance inférieure ou égale à p du message reçu, alors par inégalité triangulaire $d(m_1, m_2) \leq d(m, m_1) + d(m, m_2) \leq 2p$ donc avec notre hypothèse $(*)$ on obtient $d(m_1, m_2) < d$, ce qui par déf. de d, donne $m_1 = m_2$.

Ainsi on peut corriger un message m contenant p erreurs avec $2p \leq d-1$ i.e $p \leq \lfloor (d-1)/2 \rfloor$ en déterminant *l'unique* message du code à distance minimale de m (qui existe puisqu'on départ m a bien été fabriqué à partir d'un tel message).

En revanche si on considère deux messages m_1 et m_2 du code avec $d(m_1, m_2) = d$ alors

• si d est pair, en modifiant exactement $q = d/2$ bits de m_1, on obtient un message erroné à égale à distance de m_1 et m_2. Donc on ne sait pas comment décoder m. Or pour $d = 2q$, q bien le premier entier suivant $\lfloor (d-1)/2 \rfloor$

• si d est impair, on note $d = 2p+1$ alors $\lfloor (d-1)/2 \rfloor = p$ et dans ce cas en modifiant $p+1$ bits de m_1, on obtient un message plus proche de m_2 que de m_1.

Conclusion : on a bien montré que $\lfloor (d-1)/2 \rfloor$ est le plus grand entier p pour lequel on peut corriger p erreurs.

Q15) Ce code est fermé de deux mots $[0] * n, [1] * n$ la distance minimale est donc $d = n$. Pour $n = 3$, on retrouve bien les résultats de la Q4 : ce code permet de détecter $d - 1 = 2$ erreurs, et de corriger $\lfloor (d-1)/2 \rfloor = 1$ erreur.

Q16)
```
def distminicode(Code):
    mini=len(Code[0]) # majorant clair de la distance
    n=len(Code)
    for i in range(n-1):
        for j in range(i+1,n):
            a=dist(Code[i],Code[j])
            if a<d:
                d=a
    return(d)
```

Q17) Une telle matrice est de la forme $G = \begin{pmatrix} I_k \\ G_1 \end{pmatrix}$ où $G_1 \in \mathcal{M}n-k, k(\mathbb{F}_2)$ et I_k est la matrice unité de taille k, puisque le codage ne modifie pas les k première entrées.

Q18) Les matrices G_1 associées sont respectivement $G_1 = \begin{pmatrix} 1 & 1 & \dots & 1 \end{pmatrix}$ et $G_1' = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$.

Dans le premier cas $G = \begin{pmatrix} I_8 \\ G_1 \end{pmatrix} \in \mathcal{M}_{9,8}(\mathbb{F}_2)$ dans le second cas $G = \begin{pmatrix} 1 \\ G_1' \end{pmatrix} \in \mathcal{M}_{n,1}(\mathbb{F}_2)$.

Q19) Par déf. pour tout $m, m' \in C$ $d(m, m') = pds(m - m')$

Comme le code est linéaire, C est un e.v. donc on a l'égalité :

$$\{m - m' \text{ pour } (m, m') \in C^2, m \neq m'\} = C \setminus \{0\}$$

donc la fonction `pds` a le même minimum sur ces deux ensembles ce qui est la conclusion demandée.

Culturel : On peut démontrer (borne de Singleton, exercice !) que la distance minimale d d'un code linéaire de dimension k de longueur n vérifie l'inégalité

$$\boxed{d \leq n + 1 - k}$$

Q20) Sachant que :

$$\begin{pmatrix} G_1 & I_{n-k} \end{pmatrix} . m = G_1 \begin{pmatrix} m_0 \\ \vdots \\ m_{k-1} \end{pmatrix} + \begin{pmatrix} m_k \\ \vdots \\ m_{n-1} \end{pmatrix} = G_1 \begin{pmatrix} m_0 \\ \vdots \\ m_{k-1} \end{pmatrix} - \begin{pmatrix} m_k \\ \vdots \\ m_{n-1} \end{pmatrix}$$

puisque dans $\mathbb{F}_2$, $+1 = -1$, en notant $\tilde{m} = \begin{pmatrix} m_0 \\ \vdots \\ m_{k-1} \end{pmatrix}$ le message d'origine, nous obtenons l'équivalence :

$$Hm = 0 \Leftrightarrow G_1\tilde{m} = \begin{pmatrix} m_k \\ \vdots \\ m_{n-1} \end{pmatrix} \Leftrightarrow G\tilde{m} = m.$$

Q21)
```
def mult(M,X):
    R=[]
    for i in range(len(M)):
        S=0
        for j in range(len(M[0])):
            S=S+M[i][j]*X[j]
        S=S%2
        R.append(S)
    return R
```

A chaque tour de boucle intérieure on a un $O(1)$ opérations (une addition, une multiplication). Donc la complexité est en $O(nk)$.

Q22) On a seulement besoin pour chaque m de concaténer m avec la liste obtenue par produit avec $G1$ donc :

```
def codlinsys(G1,m):
    return(m+mult(G1,m))
```

Q23)
```
def detect(G1,m):
    k=len(G1[0]) # nb de col
    n=len(m)
    l2=[m[i] for i in range(k,n)]
    l1=[m[i] for i in range(k)]
    return(mult(G1,l1)==l2)
```

Q24) Les deux dernières questions n'ont été traitées par personne, à vous de jouer maintenant !