Exercice 1

On considère la transmission d'une séquence de bits. Tous les sept bits, on ajoute un bit—appelé *bit de parité*—égal à `0` si la somme des sept bits précédents est paire, et égal à `1` sinon.

1. Utiliser cet encodage sur la séquence : `1 1 1 1 0 1 0 1 1 0 1 0 0 1 1 1 1 0 1 0 1`.
2. Décoder la séquence : `1 1 1 0 0 1 1 1 1 1 0 1 0 1 0 1`. Quel est le problème ? Ce code permet-il de détecter n'importe quel nombre, n'importe quel type d'erreurs ?

Exercice 2

Le protocole UDP considère le contenu du message comme une séquence d'entiers sur 16 bits. Tous les segments sont ajoutés (le résultat est ajusté à 16 bits). Appelons ça la **somme**. La **somme de contrôle UDP** est calculée en tant que complément à 1 de la somme (dans le complément de 1, tous les 0 sont convertis en 1 et tous les 1 sont convertis en 0).

1. Calculez la somme de contrôle de la séquence suivante : `0110011001100110 0110011001100110 0000111100001111`.
2. Le destinataire calcule la somme et l'ajoute à la somme de contrôle de l'expéditeur. Quel devrait être le résultat ?
3. Quel est le bénéfice du protocole UDP ?

Exercice 3

Le langage Shadok contient quatre mots de base : *ga*, *bu*, *zo* et *meu*.

1. Proposer un codage binaire de longueur fixe minimal permettant de coder ces mots. Donner ses propriétés détectrices et correctrices.
2. Soit le codage binaire $\chi(ga) = \texttt{0000}$, $\chi(bu) = \texttt{0110}$, $\chi(zo) = \texttt{1001}$ et $\chi(meu) = \texttt{1111}$.
   (a) Combien d'erreurs peut-il détecter ?
   (b) Combien d'erreurs peut-il corriger ?
   (c) Si on reçoit `0000 0110 0100 1001` puis `0000 0110 0110 1001`, de quoi est-on sûr ?
3. Proposer un codage binaire 2-correcteur.

Exercice 4

Pouvoir corriger toute erreur simple pour m bits de données demande r bits de contrôle avec $m + r < 2^r$. Le codage de Hamming $(2^r - 1, 2^r - r - 1)$ prend $2^r - r - 1$ bits de données et ajoute r bits de parité aux positions $2^0, 2^1, 2^2, \ldots, 2^{r-1}$. Le bit de donnée en position $\sum_{j=0}^{r-1} a_j 2^j$ est vérifié par les bits de parité aux positions j avec $a_j = \texttt{1}$. Ainsi, le bit en position 2^0 est le bit de parité de ceux aux positions $3, 5, 7, 9, 11, 13, \ldots$, le bit en position 2^1 est le bit de parité de ceux aux positions $3, 6, 7, 10, 11, 14, 15, \ldots$ Si un bit est erroné, sa position est obtenue comme la somme des positions des bits de parité non conformes.

0. Justifier la première phrase de l'énoncé.
1. Compléter le mot | | | 1 | | 0 | 1 | 0 | en un mot du code de Hamming (7,4).
2. Les mots `1010110` et `0110011` se présentent au destinataire pour le code de Hamming (7,4). En supposant qu'il n'y ait pas plus d'un bit erroné par mot, retrouver les mots d'origine.
3. Même question avec les mots `0x3fa4`, `0x5d4b` et `0x046d` pour le code de Hamming (15,11).

Nous allons travailler avec le code RSA. Pour cela, il faut définir une classe Java `RSA` dans un fichier `RSA.java` contenant les instructions suivantes.

```
public class RSA {
  public static void main(String[] args) {
  }
}
```

Exercice 1

Écrire une fonction Java `getN` qui prend deux entiers premiers p et q tels que $2 < p, q$ et renvoie le produit $n = p \times q$. Par exemple, `getN(13,17)` renvoie `221`.

Exercice 2

Écrire une fonction Java `getPhi` qui prend deux entiers premiers p et q tels que $2 < p, q$ et renvoie le produit $(p-1) \times (q-1)$. Par exemple, `getPhi(13,17)` renvoie `192`.

Exercice 3

Écrire une fonction Java `pgcd` qui prend deux entiers a et b tels que $0 < a \leq b$ et renvoie le plus grand commun diviseur de a et b. Par exemple, `pgcd(20, 30)` renvoie `10`. Pour implémenter la fonction, nous vérifions s'il existe un entier i compris entre 1 et a qui divise a et b et nous renvoyons le plus grand i trouvé.

Exercice 4

Écrire une fonction Java `getE` qui prend un entier ϕ tels que $3 < \phi$ et renvoie un entier $e > 1$ tel que le plus grand commun diviseur de e et ϕ est 1. Par exemple, `getE(192)` peut renvoyer `5`. Pour implémenter la fonction, on cherche un entier e compris entre 2 et $\phi - 1$ tel que `pgcd(e, phi)` renvoie 1.

Exercice 5

Écrire une fonction Java `getD` qui prend deux entiers e et ϕ tels que $1 < e < \phi$ et renvoie un entier $d > 1$ tel que $(e \times d) \pmod{\phi} = 1$. Par exemple, `getD(5, 192)` renvoie `77`. Pour implémenter la fonction, on peut essayer avec tous les entier d compris entre 2 et $\phi - 1$ et nous nous arrêtons quand nous trouvons un entier d tel que $(e \times d) \pmod{\phi} = 1$.

Exercice 6

Écrire une fonction Java `modularPower` qui prend trois entiers b, e et m tels que $0 \leq b < m$ et $0 \leq e$ et renvoie $(b^e) \pmod{m}$. Par exemple, `modularPower(71,5,221)` renvoie `158`. Pour implémenter la fonction, on peut l'algorithme vu en amphi, c'est-à-dire, on peut utiliser la méthode des multiplications modulaires successives vue en amphi.

Exercice 7

En utilisant la fonction `modularPower`, écrire une fonction Java `encode` qui prend trois entiers m, n et e tels que $0 \leq m < n$ $1 < e$ et renvoie le chiffrage RSA de m avec la clé publique (n, e). Par exemple, `encode(72,221,5)` renvoie `89`.

Exercice 8

En utilisant la fonction `modularPower`, écrire une fonction Java `decode` qui prend trois entiers c, n et d tels que $0 \leq c < n$ $1 < d$ et renvoie le dechiffrage RSA de c avec la clé privée (n, d). Par exemple, `decode(89,221,77)` renvoie `72`. Vérifier que, effectivement, la fonction `decode` est l'inverse de la fonction `encode`.

Les exercices suivants modifient les fonctions `pgcd` et `modularPower` pour les rendre plus efficaces.

Exercice 9

Écrire une nouvelle version de la fonction `pgcd`, en utilisant le théorème d'Euclide qui dit que $\text{pgcd}(a, b) = \text{pgcd}(r, a)$ où r est le reste de la division de b par a. Donc, l'algorithme d'Euclide sur deux nombres entiers positifs a et b avec $a \leq b$ procède comme suit. L'algorithme regarde si $a = 0$ ou non. Si $a = 0$, le plus grand commun diviseur est la valeur b. Sinon, l'algorithme calcule le reste r de la division de b par a et recommence mais avec $a = r$ et $b = a$.

Exercice 10

Écrire une nouvelle version de la fonction `modularPower`, en utilisant l'algorithme vu en amphi suivant sur trois entiers b, e et m tels que $0 \leq b < m$ et $0 \leq e$. Initialement, l'algorithme définit la valeur r égale à 1. Après, l'algorithme regarde si $e = 0$ ou non. Si $e = 0$, l'algorithme renvoie r. Sinon, si e est impaire alors r devient $r * b$. L'algorithme recommence mais avec $e = e/2$ et $b = b^2$.