1. Pseudozufallszahlengenerierung

Implementieren Sie in JAVA einen Pseudozufallszahlengenerator nach der Linearen Kongruenzmethode!

a. Stellen Sie eine Klasse **`LCG`** mit einer Methode **`int nextInt()`** zur Verfügung. Diese Methode soll nach der linearen Kongruenzmethode einen „Zufallswert" liefern. Der Startwert des Pseudozufallszahlengenerators (X_0) soll dem Konstruktor der **`LCG`** – Klasse als Parameter übergeben werden.

- *Verwenden Sie eine Parameterkombination für a, b und N aus der Datei „LinearerKongruenzgenerator-Infos.pdf"!*
- *Achten Sie grundsätzlich auf die Verwendung eines geeigneten Datentyps (z.B. „long") für interne Berechnungen, um mögliche Überläufe zu vermeiden!*

b. Testen Sie Ihre Implementierung, indem Sie für 256 durch LCG erzeugte Pseudozufallszahlen (Startwert beliebig) jeweils das niederwertigste Byte (0-255) ausgeben lassen!

- *Das niederwertigste Byte einer int-Variablen* **`value`** *erhalten Sie durch den Ausdruck* **`value & 0x000000FF`**

2. Stromchiffre

a. Schreiben Sie unter Verwendung der **`LCG`** – Klasse aus Teil 1 ein JAVA-Programm **`HC1`** („HAW-Chiffre 1"), welches als Eingabeparameter von der Standardeingabe einen numerischen Schlüssel (Startwert) sowie den Pfad für eine zu verschlüsselnde / entschlüsselnde Datei erhält.
Ihr Programm soll jedes Byte der Datei mit einem – ausgehend vom übergebenen Startwert – „zufällig" erzeugten Schlüsselbyte mittels XOR verknüpfen und das Ergebnis in eine neue Chiffredatei ausgeben.

- *Arbeiten Sie mit Input/Outputstreams und vermeiden Sie die Verwendung von „Buffered Reader" oder „Buffered Writer" – Klassen!*

b. Testen Sie Ihre Stromchiffre **`HC1`**, indem Sie eine Klartextdatei verschlüsseln und die erzeugte Chiffredatei anschließend durch einen nochmaligen Aufruf von **`HC1`** wieder entschlüsseln.
Verifizieren Sie (z.B. mittels „diff"), dass beide Dateien identische Inhalte besitzen.

c. Ersetzen Sie die Klasse **`LCG`** in Ihrem **`HC1`**-Programm durch die Klasse **`java.security.SecureRandom`** (die von **`java.util.Random`** abgeleitet ist) und wiederholen Sie den Test unter b.

3. TripleDES als Blockchiffre

a. Implementieren Sie in JAVA ein Programm **`TripleDES`** mit der Aufgabe, alle Bytes einer beliebigen Datei mit dem TripleDES-Verfahren zu verschlüsseln oder zu entschlüsseln. Ihr Programm soll mit drei verschiedenen DES-Schlüsseln arbeiten und in der Reihenfolge EDE (encrypt-decrypt-encrypt) das DES-Verfahren dreimal hintereinander durchlaufen. Implementieren Sie als Blockchiffre-Betriebsart[1] den **CFB**-Modus („Cipher Feedback"-Modus).

Kommandozeilen-Parameter:

1. Dateiname einer zu verschlüsselnden/entschlüsselnden Datei
2. Dateiname einer Schlüssel-Datei mit folgendem Inhalt:
   Byte 1-24: 24 Schlüsselbytes (3 DES-Schlüssel mit je 8 Byte, wobei von jedem Byte jeweils 7 Bit verwendet werden)
   Byte 25-32: 8 Bytes für den Initialisierungsvektor zum Betrieb im CFB - Modus
3. Dateiname der Ausgabedatei
4. Status-String zur Angabe der gewünschten Operation:
   `encrypt` – Verschlüsselung der Datei
   `decrypt` – Entschlüsselung der Datei

b. Testen Sie Ihre Implementierung, indem Sie die Datei **`3DESTest.enc`** entschlüsseln (Tipp: Ergebnis ist eine PDF-Datei). Schlüssel und Initialisierungsvektor finden Sie in der Datei **`3DESTest.key`**.

- *Benutzen Sie als DES-Implementierung die mitgelieferte Datei `DES.java` !*

- *Beispielcode zum blockweisen Kopieren einer Datei:*

```
FileInputStream in;
FileOutputStream out;
byte[] buffer = new byte[8];
int len;
while ((len = in.read(buffer)) > 0) {
     out.write(buffer, 0, len);
}
```

Viel Spaß!

[1] Die Betriebsart bezieht sich hier auf das gesamte TripleDES-Verfahren, nicht auf die einzelnen DES-Aufrufe!