Challenge 1: LFSR

A common technique for obfuscating data is to use exclusive-or (XOR) with some key; it is inexpensive and symmetric. A problem occurs when this is used on file formats like portable executable where there are many null bytes, since XORing nulls with the key ends up writing the key out.

A slightly more complex algorithm uses a Linear Feedback Shift Register (LFSR) to produce a key stream, which will be XORed with the data. A generic LFSR is:

> If F is the value which represents the LFSR feedback and S is the current state of the LFSR, the next state of the LFSR is computed as follows:
>
> if the lowest bit of S is 0: `S = S >> 1`
>
> if the lowest bit of S is 1: `S = (S >> 1) ^ F`

For this challenge, you'll be using an LFSR with the feedback value 0x87654321. The LFSR is initialized with a value and stepped to produce the key stream. The next key value is read from the LFSR after eight steps, with the actual key being the lowest byte of the current LFSR value.

For example, if the initial value of the LFSR is `0xFFFFFFFF`, then next value after stepping it eight times will be `0x9243F425`, meaning that the first key byte is `0x25`. If the first byte of the input is 0x41, the first byte of output will be 0x64.

Your task is to implement this algorithm (both LFSR and the XOR). We're only interested in algorithm implementation; other code will be discarded.

The function should adhere to one of following signatures:

C/C++	`unsigned char *Crypt(unsigned char *data, int dataLength, unsigned int initialValue)`
C#	`static byte[] Crypt(byte[] data, uint initialValue)`
Python	`Crypt(data, initialValue) # returns byte string`
Java	`static byte[] Crypt(byte[] data, long initialValue)`

Example Tests:

data	“apple”
dataLength	5
initialValue	0x12345678
result	“\xCD\x01\xEF\xD7\x30”

data	“\xCD\x01\xEF\xD7\x30”
dataLength	5
initialValue	0x12345678
result	“apple”

Submit: Source code containing your implementation of the LFSR based encoding routine.

Challenge 2: KDB Files

During a forensic investigation, a new data storage file format is discovered with the extension KDB. In addition to being stored in a custom format, the contents are encoded with the LFSR algorithm seen in Challenge 1, so all of the data contained in the file format must also be decoded once extracted. Using the provided spec on the next page, create a program that extracts and decodes the enclosed data.

- Your program should accept a path to a KDB file via command line argument.
- Your program should report the extracted, decoded information to standard out with each entry (name and stored information) on a separate line.

Included is a sample KDB file named "store.kdb". The initial value for the LFSR algorithm is 0x4F574154.

Submit: The source code of your solution **AND** a text file containing the standard out when your solution is run with store.kdb as its input.

KDB File Specification

HEAD		
MAGIC	BYTE[6]	"CT2018"
ENTRY_LIST *	INT32	Pointer to the ENTRY_LIST

ENTRY_LIST		
ENTRIES	ENTRY[127]	Array of ENTRIES

ENTRY		
NAME	CHAR[16]	Null terminated string
BLOCK_LIST *	INT32	Pointer to the BLOCK_LIST

BLOCK_LIST		
BLOCKS	BLOCK[255]	Array of BLOCKS

BLOCK		
SIZE	INT16	Length of the BLOCK's data
DATA *	INT32	Pointer to the BLOCK's data

DATA		
DATA	BYTE[]	An array of bytes

- All pointers are relative to the beginning of the file.
- The BLOCK's size and all pointers are little endian.
- All KDB files begin with a HEAD at 0x0, which starts with the magic bytes "CT2018" and contains a pointer to the file's ENTRY_LIST.
- The ENTRY_LIST is an array of up to 127 ENTRIES. The value 0xFFFFFFFF signifies the end of the ENTRY_LIST.
- Each ENTRY has an ASCII encoded, null terminated name and a pointer to the ENTRY's BLOCK_LIST.
- The BLOCK_LIST is an ordered array of BLOCKS. The value 0xFFFFFFFF signifies the end of the BLOCK_LIST.
- Each BLOCK contains a pointer to that BLOCK's DATA and the size of that DATA.
- The DATA is an LFSR (from Challenge 1) encoded byte array. All BLOCKS from an ENTRY are combined then decoded to reveal the stored information.

Challenge 3: Obfuscated JPEGs

During the same forensic investigation, multiple obfuscated JPEGs were discovered. All JPEGs start with the bytes FF D8 FF, but in this case, these bytes were changed to prevent applications from recognizing the files as JPEGs. All of these JPEGs are guaranteed to end with the standard magic bytes FF D9.

The investigation requires that these JPEGs be identified and the initial bytes repaired so they can be viewed. Identifying the JPEGs should be a trivial task, but none of our tools are set up to recognize the custom header. Create a tool that, given an input file, can detect, extract, repair, and save these obfuscated JPEGs. This process is commonly known as file carving (with simple file repair).

- Your program should accept a path to a KDB file followed by another file path via command line arguments.
- Your program should load the magic bytes from the KDB file's entry named "MAGIC" using the algorithm from Challenge 2. This file does not contain any obfuscated JPEGs.
- Your program should detect all JPEG files with the custom magic bytes that are present in the second provided file.
- Your program should **NOT** attempt to detect JPEGs not starting with the custom magic bytes.
- Your program should replace the custom magic bytes from the KDB with the standard ones (FF D8 FF).
- Your program should output all repaired JPEGs to a directory "<inputFileName>_Repaired" in the same directory as the program binary.
- Your program should save the repaired JPEGs with the filename "<offset>.jpeg".
- Your program should report to standard out the offset at which it detected each file, the file's size, the file's MD5 hash (after it is repaired), and the path of the corresponding output file.
- Your program should **NOT** attempt to OCR the repaired JPEGs to find the hidden message; just view them yourself.

Your solution to this challenge may not make use of any existing library for parsing JPEG files, file carving, file typing, or steganography. You may (and will probably want to) use an existing MD5 implementation.

Included is a sample input file named “input.bin” that contains three valid JPEGs and the corresponding KDB file “magic.kdb”.

Submit: The source code of your solution **AND** a text file containing the contents of standard out when your solution is run with magic.kdb and input.bin as its input **AND** a text file containing the hidden messages in the repaired JPEGs. Do **NOT** submit the repaired JPEGs.