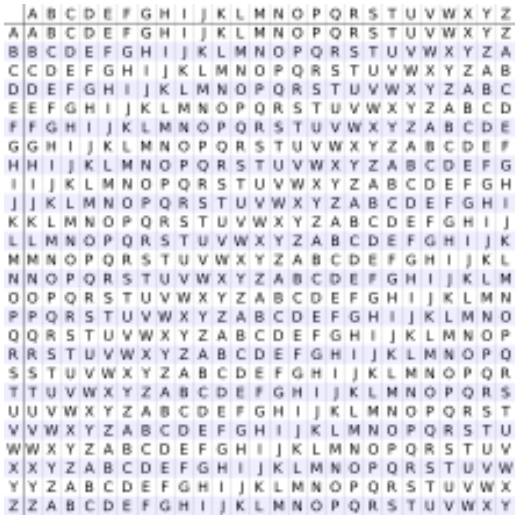
CS 419 Homework 10 (project 3)

Due Wednesday, November 4, 2020 11:59pm

Part 1: Binary Vigenère Cipher



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

We covered the cipher in class. It was designed for manual cryptography (pencil and paper) and used a repeating key on a grid.

Each row contains the alphabet shifted by one position. To encrypt, you:

1. Find the row indexed by the plaintext letter.
2. Find the column indexed by the next character of the key
3. The ciphertext letter is the intersection.

The key is repeated to make it the same length as the message.

Your program will create a binary version of this cipher.
The key can be either text or, preferably, binary data and is stored in a key file.

The usage of the command is:

```
vencrypt keyfile message ciphertext
```

This encrypts the *message* file using the key in *keyfile* and produces the file *ciphertext*.
Here's what you need to do:

1. Create a 256x256 grid.
   a. Row 0 contains 00 .. 0xff
   b. Row 1 contains 01 .. 0xff, 00
   c. Row 2 contains 02 .. 0xff, 00, 01
   d. Row 255 contains 0xff, 00 .. 0xfe

2. To encrypt a byte of plaintext:
   a. Look up table
      `ciphertext[n] = [`*`row=message[n]`*`][`*`column=ciphertext[i]`*`]`
   b. `n = n+1`
   c. `i = (i+1) % length(ciphertext)`

If you think about the operations that take place, you will discover that you don't even need a table since you can easily derive the value of the ciphertext from the key byte and plaintext byte.

Then, create a decryption function

```
vdecrypt keyfile ciphertext message
```

This reads the key from *keyfile* and decrypts the contents in *cipherfile* into the file *message*.

Part 2: Stream cipher

We also covered stream ciphers in class. These simulate the one-time pad by using a keystream generator to create a keystream that is the same length as the message.

The keystream generator is simply a pseudorandom number generator and the seed is derived from the password. You will always see the same sequence of numbers for the same seed.

To implement this cipher, you will:

1. Implement a linear congruential generator. It is a trivial formula that is described here: https://en.wikipedia.org/wiki/Linear_congruential_generator

   Implementing it is only three lines of code!
   Use these parameters as described in the article:
   Modulus = 256 (1 byte)
   Multiplier = 1103515245
   Increment = 12345

   By using a well-known formula, your output should be the same regardless of the programming language or operating system you use.

2. Convert the password to a seed using the C code here (may need to convert it to python or java):
   http://www.cse.yorku.ca/~oz/hash.html
   This is also three lines of code!
   As with the previous step, this implementation should ensure that your output will be the same regardless of the programmer, programming language, or operating system.

3. Apply the stream cipher. The ciphertext is generated byte by byte and is simply:
   $ciphertext_i = plaintext_i \oplus keytext_i$

Because applying an exclusive-or of the same key a second time undoes the first exclusive-or, you only need to implement one command.

Usage:

```
scrypt password plaintext ciphertext
scrypt password ciphertext plaintext
```

The *password* is a text string. The parameters *plaintext* and *ciphertext* are files. The same program can be used to encrypt or decrypt.

Part 3: Block encryption with cipher block chaining and padding

This is an enhancement of Part 2 to use the concepts of shuffling data, padding – adding it and removing it correctly, and cipher block chaining.

We modify the stream cipher above to have it operate on 16-byte blocks instead of bytes. This turns it into a form of block cipher. A block cipher uses multiple iterations (rounds) through an SP-network (substitutions & permutations) to add confusion & diffusion. Confusion refers to changing bit values as a function of the key and that each bit of the ciphertext is determined by several parts of the key. Diffusion refers to the property that a change in one bit of plaintext will result in many bits of the ciphertext changing (about half).

We will not use multiple rounds of an SP network. Instead, we will keep the mechanisms of the stream cipher in place but enhance it in two ways:

1. Confusion is roughly determined by the seed and the pseudorandom output of the keystream generator in our case, but we will enhance the degree confusion by shuffling bytes of the block.

   We will have the key determine which sets of bytes in the block to exchange (swap). For each 16-byte block, do the following:

   ```
   for (i=0; i < blocksize; i=i+1)
         first = key[i] & 0xf  (lower 4 bits of the keystream)
         second = (key[i] >> 4) & 0xf   (top 4 bits of the keystream)
         swap(block[first], block[second])      (exchange the bytes)
   ```

2. Stream ciphers have no (or low) diffusion. The change of a bit in plaintext will generally affect only that bit in ciphertext. We will enhance diffusion by adding cipher block chaining (CBC). With cipher block chaining, we exclusive-or the previous block with the next block.

The flow of the cipher is:

> Create an initialization vector (IV) by reading 16 bytes from the keystream generator.
> For each 16-byte *plaintext_block*$_N$:
> 1. If it is the last block, add padding.

2. Apply CBC: $temp\_block_N = plaintext\_block_N \oplus ciphertext\_block_{N-1}$
   Use the initialization vector this is the first block.
3. Read 16 bytes from the keystream.
4. Shuffle the bytes based on the keystream data.
5. $ciphertext\_block_N = temp\_block_N \oplus keystream_N$.
6. Write $ciphertext\_block_N$.

You need to:

1. Use the same password hashing and keystream generator in Part 1.
2. Use an initialization vector for the first block. This will be the first 16 bytes from the keystream generator.
3. Add padding. This will be an amount from 1 through 16 bytes
   (e.g., finish up a block or add a new block). The padding is added before any encryption or shuffling takes place.

   Each byte of the padding will contain the number of bytes of padding that were added. For example, if three bytes were added, then the last three bytes of the file will be 3, 3, 3.

   This is a technique that allows you to know how much padding needs to be removed when decrypting and writing the plaintext output.

You will need to write two programs for this part, one to encrypt and another to decrypt:

```
sbencrypt password plaintext ciphertext
sbdecrypt password ciphertext plaintext
```

As with Part 2, the program will take a *password* string, which will be hashed and used as a seed for the keystream generator. The parameters *plaintext* and *ciphertext* are both file names.

Reference programs

It's important that encryption software works consistently across multiple systems regardless of author, programming language, or operating system. I should be able to encrypt a message on my Mac and expect you to be able to decrypt it with your program on your Raspberry Pi running Linux.

You are provided with reference versions of the programs that you can use to compare with yours and, perhaps, help debug your code. Here are the programs provided:

Binary Vigenère Cipher

```
vencrypt [-d] [keyfile | -k key] plaintext ciphertext
```

Encrypt a *plaintext* file to create a *ciphertext* file using key data stored in *keyfile*. Alternatively, you can specify a textual keyfile with the -k option. For example,

```
        vencrypt -k monkey01 file.txt file.enc
```

Will use the bytes in “monkey01” as the key. This limits the key space but may be useful for debugging.

The `-d` flag turns on debugging information and shows you what lookups are taking place (note that you don’t need to implement a table but can do so if you find that easier).

Stream Cipher – keystream test

Before you test your cipher, make sure that your password hash and pseudorandom number generator are producing the proper results. You can test this with the *prand-test* program:

```
        prand-test [-p password | -s seed] [-n num]
```

If the program is supplied a password with the `-p` parameter, the password will be hashed and the result shown.

The `-n` parameter lets you specify the number of pseudorandom numbers to be printed. The default is 0.

If you just want to see the list of pseudorandom numbers generated from a specific seed, you can specify a seed number instead of a password with the `-s` parameter.

Stream Cipher

```
        scrypt [-d] password plaintext ciphertext
```

Encrypt a *plaintext* file into a *ciphertext* file using a keystream derived from the *password* string. The same command decrypts a *ciphertext* file into a *plaintext* file.

```
        scrypt [-d] password ciphertext plaintext
```

The `-d` flag turns on debugging mode and shows the series of xor operations from the source file to the output file.

Block Cipher

```
        sbencrypt [-d] password plaintextfile ciphertextfile
```

Encrypts a *plaintext* file into a *ciphertext* file using a keystream derived from the *password* string. The same command decrypts a *ciphertext* file into a *plaintext* file.

```
sbdecrypt [-d] password ciphertextfile plaintextfile
```

Decrypts a *ciphertext* file into a *plaintext* file using a keystream derived from the *password* string. The same command decrypts a *ciphertext* file into a *plaintext* file.

For both commands, the `-d` flag enables debugging, showing the sequence of shuffling per block.