Aaron Fox
CECS 564-01
Dr. Desoky

Project 2: Content Scrambling System Cipher

Introduction: An implementation of the Content Scrambling System (CSS) cipher, originally devised in 1996, was successfully carried out in Project 2, following the specifications given in the project prompt. `The file Fox_Project_2_CSS.py` was written in Python. It allows for the encryption and decryption of any binary file that is input from a local file on one's computer. The encryption uses two linear-feedback shift registers (LFSR) to generate pseudorandom numbers based on a seeded 40-bit key combined with a full adder to add the result of the pseudorandom numbers generated from those LFSRs. The result of the full adder is then XORd bitwise with the input characters (padding is added to make the inputs of size 8-bits as needed.) The result of XORing each character of the text to encrypt and the full adder result is then converted to its decimal equivalent which is then converted to its equivalent Unicode value that is then written to an output binary file. The decryption of the files is also written with a very similar method as that of the encryption, for the decrypting can be accomplished by simply encrypting the encrypted text again with the same key since the encryption keys are always involutory since `XOR(XOR(key, text), key) = text` per the boolean logic of XOR.

A report was then written discussing the results and methods of designing the CSS cipher and the statistical changes of the encrypted text vs the unencrypted text.

Encrypting: The main CSS encryption took place in the `encrypt_css` function. This function took in two parameters: `key` and `text_to_encrypt`. `key` consists of the 40-bit key given as a list of five bytes ranging from 0-255, such as `[25, 230, 3, 64, 12]`, which is equivalent to `'0001100111100110000000110100000000001100'` in binary. `text_to_encrypt` is the plaintext string of the text to encrypt. The user must first read in the text to the string variable from the input file before sending it into the `encrypt_css`, like so:

```
                        # Encrypting text
                        text_file_path = r'C:\Users\aaron\Classes_11th_Semester\CECS
564\CECS-564-Cryptography\Project 2\helloworld.txt'
                        file_to_encrypt = open(text_file_path, 'r', encoding='latin1')
                        text_to_encrypt = file_to_encrypt.read()
                        file_to_encrypt.close()
```

After the user inputs these two parameters to `encrypt_css`, the function then executes the cipher functionality. encrypt_css begins by finding the 40-bit binary key to be used for seeding the LSFRs from the given key. The function does this by assigning the 40-bit string `binary_40_bit_key` to the value output by the method `convert_bytes_to_binary_number`, which takes in the parameter `key`. `convert_bytes_to_binary_number` e iterates over every byte, finds its binary equivalent, pads the number with 0s to make sure each number has 8 bits (e.g. the decimal number 3 would be 00000011 in binary while the decimal number 243 is 11110011 is

binary, so both are 8 bits.) This ensures that all 5 numbers in the given input key list yields exactly one byte. This makes sure that the returned key is 40 bits (5 bytes) long. It's code is found below:

```
def convert_bytes_to_binary_number(bytes):
    binary_40_bit_string = ''
    for byte in bytes:
        binary = str(bin(byte))[2:]
        # Add padding of 0's to binary if needed
        binary = '0' * (8- len(binary)) + binary
        binary_40_bit_string = binary_40_bit_string + binary
    print("converted 40-bit key is " + binary_40_bit_string)
    return binary_40_bit_string
```

The initialization keys for each LFSR were then assigned such that the first 16 bits of the binary string, plus one default 1 at the beginning of the bits, (hence why each primitive polynomial ends in +1 in the prompt) are one. This ensures that the first LFSR always is 17-bits long, per the CSS specifications. The initialization key for the second LFSR are initialized with the final 24 bits found in `binary_40_bit_key` and then a 1 bit is again added as the last bit to the second LFSR, ensuring that the second LFSR is 25-bits long per the CSS specifications.

The actual LSFRs themselves are then instantiated using Python generators, and they are instantiated as such:

```
LFSR_1_generator = LFSR_generator((15, 1), intialization_key_for_LSFR_1)
LFSR_2_generator = LFSR_generator((15, 5, 4, 1), intialization_key_for_LSFR_2)
```

The generator function `LFSR_generator` took in two parameters, `taps` and `seed`. `taps` represents the primitive polynomial which served as taps for the XOR function that fed into the most significant bit. For example, the prompt of $C_1(x) = x^{15} + x + 1$ for the first LFSR is translated to the list of (15, 1) above (the + 1 is implicit and automatically added by default into the LFSR from previous code.) `seed` represents the initialization keys derived from the `binary_40_bit_key` as described above.

The code for the LFSR generator is seen below:

```
def LFSR_generator(taps, seed):
    s = seed
    xor_output = 0
    yield s[len(s)-1]
    while 1:
        for tap in taps:
            xor_output = xor_output + int(s[len(s)-tap])
        if xor_output % 2 == 0.0:
            xor_output = 0
        else:
            xor_output = 1
```

```
            s = str(xor_output) + s[0:len(s) - 1]
            xor_output = 0
            yield s[len(s)-1]
```

This function uses a Python generator, which, instead of returning a variable each time its called, simply yields a variable one at a time. The generator saves the state of all the local variables and so will continually yield the next shifted bit for each register when they are needed by calling `next(LFSR_1_generator)` when the next shifted bit of each register is needed. In place of actually XORing all the given taps for the generator, the XORs are all added together and then modulus 2'd. If the result modulus 2 produces 0, then the xor_output is 0; else, it is 1. This effectively shifts the bits by one each time the `next` function is called on the generator.

An output file to write the text to is then opened for writing the results of encrypting the text per the CSS specifications. Afterward, every character in the input parameter `text_to_encrypt` is then iterated over in a for loop and 8 binary bits from each LFSR are then taken placed into the respective variables of `register_1_binary` and `register_2_binary` using their respective generator functions to obtain the shifted bits, like so:

```
        # Get 8 binary bits
        register_1_binary = ''
        register_2_binary = ''
        for i in range(8):
            register_1_binary = register_1_binary + str(next(LFSR_1_generator))
            register_2_binary = register_2_binary + str(next(LFSR_2_generator))
```

Each binary result generated above is then put through the 8-bit full adder and stored in the variable `full_adder_result`. This is accomplished in the line of code,

```
full_adder_result = x_bit_full_adder(register_1_binary, register_2_binary)
```

The function `x_bit_full_adder` takes in two parameters that are the binary strings which should be of the same size (in this particular project, two 8-bit binary strings are used.) The code for the X-bit full adder is as follows:

```
def x_bit_full_adder(bits_1, bits_2):
    # Initial carry bit is 0 per CSS Project 2 prompt
    carry_bit = 0
    # Store results into string backwards so that it can be reversed later
    result = ''
    for i in range(len(bits_1) - 1, -1, -1):
        sum_bit, carry_bit = full_adder(int(bits_1[i]), int(bits_2[i]), carry_bit)
        result = result + str(sum_bit)
    # Return result but inverse
    return result[::-1]
```

The function works by combining X amount of full adders with the initial carry bit as zero. It iterates from the last bit of each string and places the result of the `full_adder` and changes the carry bit and then iterates forward. The code for `full_adder` is seen below:

```
def full_adder(bit_1, bit_2, carry_bit=0):
    sum_1, carry_1 = half_adder(bit_1, bit_2)
    sum_2, carry_2 = half_adder(sum_1, carry_bit)

    return (sum_2, carry_1 or carry_2)
```

This code works by using two half adders and some basic boolean logic of combining the two half adders. The code for the half_adders is given below:

```
def half_adder(bit_1, bit_2):
    return (bit_1 ^ bit_2, bit_1 and bit_2)
```

where the ^ is the bitwise XOR operator. Combining all of this boolean logic into the full adder allows for the stacking of multiple full adders so that any amount of bits can be added together, including, of course, the needed 8 bit strings that are output by the LFSRs.

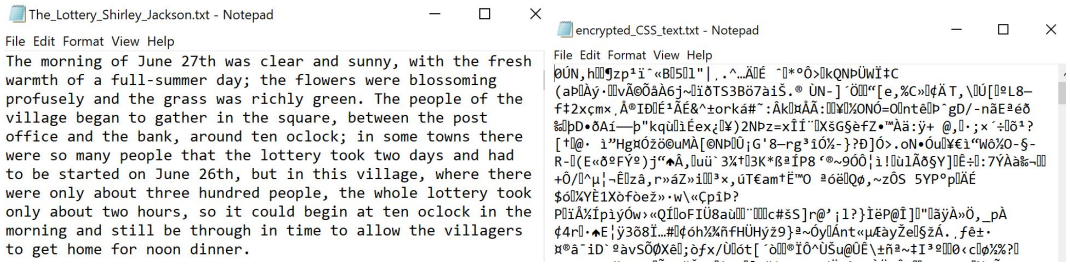
The `full_adder_result` of the two binary strings are then XORd with the binary representation of the Unicode value of the character of the text to be encrypted using the `xor_binary` function. The `xor_binary` function works as such:

```
def xor_binary(binary_1, binary_2):
    # Make sure both binary strings are of length 8 by padding them with 0s at
    # beginning if necessary
    binary_1 = '0' * (8 - len(binary_1)) + binary_1
    binary_2 = '0' * (8 - len(binary_2)) + binary_2

    result = ''
    for i in range(len(binary_1)):
        result = result + str((int(binary_1[i]) ^ int(binary_2[i])))

    return result
```

The function works by first padding the beginning of each input binary string with 0’s on the left, (normally just one 0 is needed for `binary_2` since most plaintext letters and punctuation in the English language only need 7 bits to represent their values.

The result of XORing the result of the 8-bit full adder and the character from the input plaintext is then converted to its equivalent decimal value. This decimal value is then converted to its equivalent unicode value and then written to the output encrypted file.

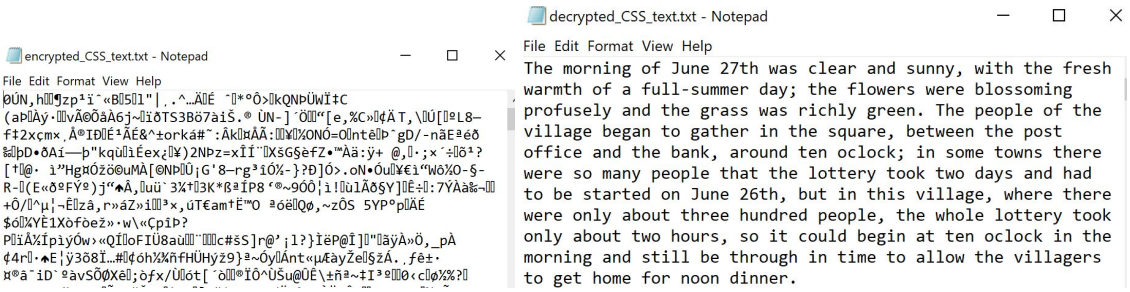
The results of encrypting the binary file can be seen below:



Before (left) and after (right) encrypting the binary file

Decryption: Since (XOR(XOR(key, text), key) = text per the boolean logic of XOR), the key is always involutory. This means that encryption(decryption(X)) = encryption(encryption(X)) = X. This means that encrypting something again with the same key will change the encrypted text back to its original text. The code for writing the decryption of text is thus written almost exactly the same as the encryption function, barring the name changes of encrypt to decrypt and the change of file output (now named `decrypted_CSS_text.txt` instead). The code for `decrypt_CSS(key, text_to_decrypt)` can be found in the Appendix of this text file.

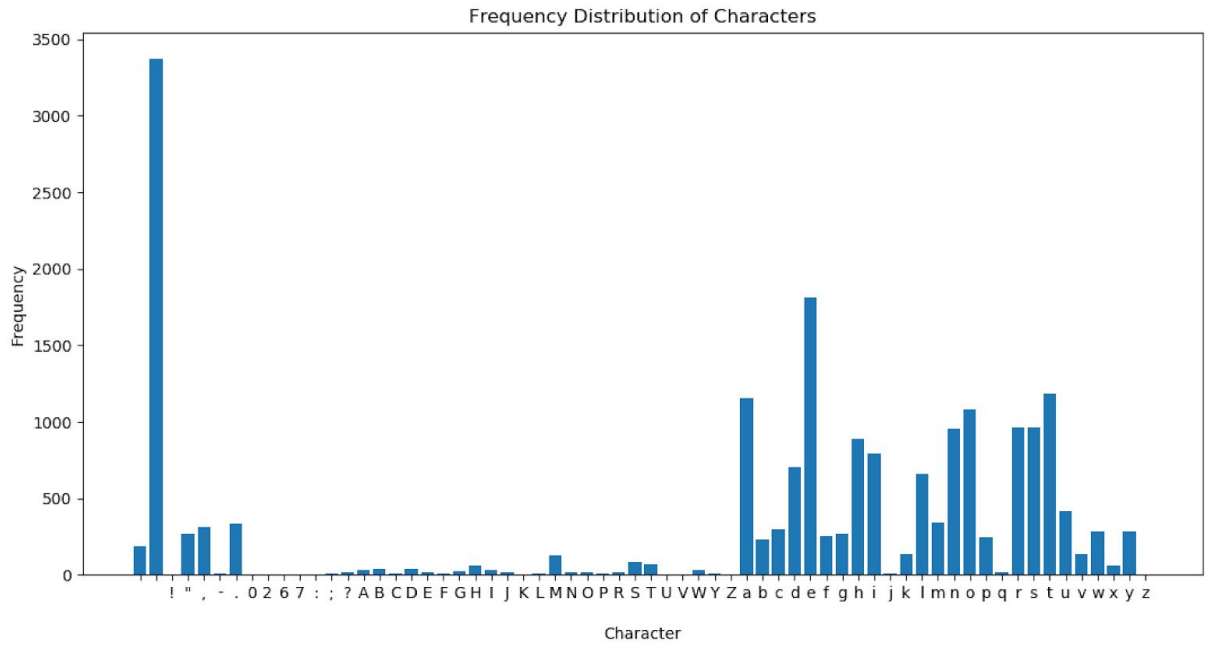
The results of decrypting the binary file can be seen below:



Before (left) and after (right) decrypting the encrypted text

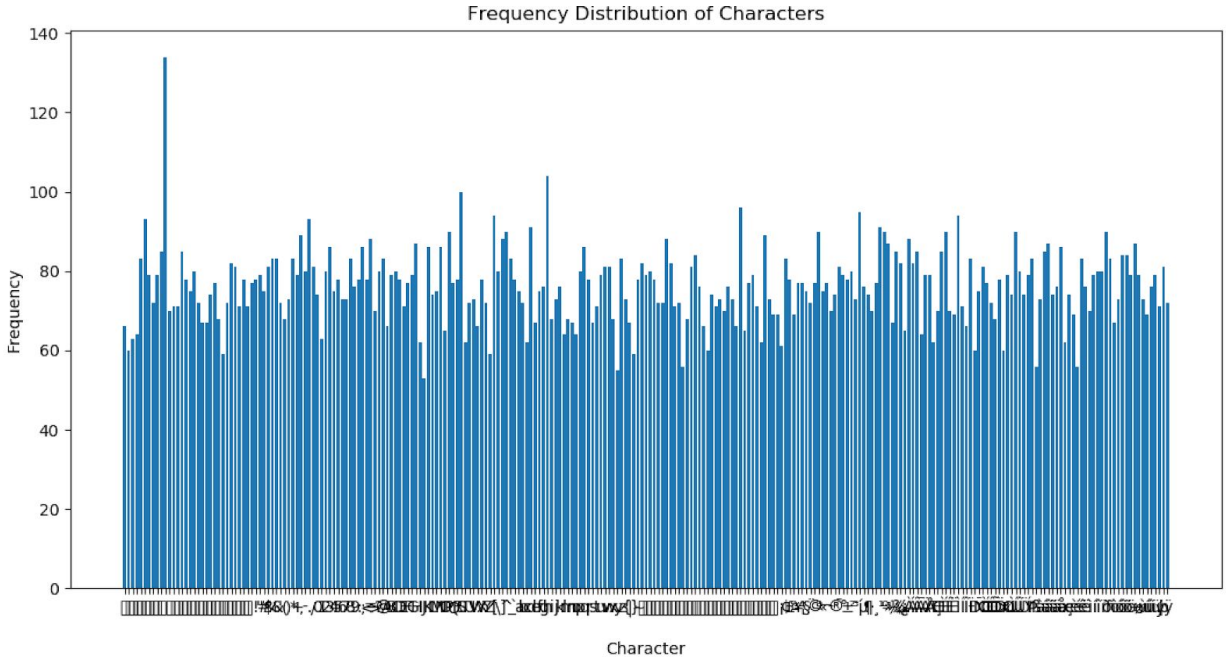
Statistical Changes:
Data statistics: What is the effect of CSS encryption on the data statistics of .txt data such as mode, mean, median, standard deviation and entropy?

Below is the frequency distribution of the unencrypted text of the short story *The Lottery*. This is the common English language distribution of unicode characters, with the most popular letters, in order, being the space key, the letter e, and then the letter t.



Frequency Distribution of unencrypted English data

After running the data through the CSS encryption, it can be seen that the data varies wildly from Unicode characters varying from characters representing the numbers 0-255. The graph of the frequency distribution can be seen below:



Frequency Distribution of the Encrypted English data using the CSS algorithm

It can be seen that the encrypted data frequency distribution appears very random and, not knowing the key, would require something along the lines of a brute-force attack to successfully

crack. Because the encrypted text has effectively every single possible unicode letter from 0-255 because of the large 40-bit key, there is no easily discernible pattern from any cycles or codes.

The mean of the unencrypted original text of the short story was 89.2091, around the letter Y and Z in the Unicode alphabet. This makes sense because of the balance between the many spaces and lowercase letters would lead the average to be somewhere in between the two at the capital letters.

The mean of the text that has been encrypted using the CSS cipher is 127.7936, around the delete and control characters in the Unicode text. This number makes sense, for it is around exactly half of the numbers between 0 and 255. Since the numbers are effectively randomly distributed between 0-255 in an almost uniform manner due to the LFSR's pseudo-random nature, the mean of the text value appears to correctly lie around halfway in the Unicode text from 0-255.

The standard deviation for the unencrypted plaintext file is 32.1314, which is typical for a standard English text, which often range from around 30 to 34 based on the large English language files tested (e.g. Darwin.txt had a standard deviation of 30.04604.)

The standard deviation for the text encrypted using the CSS cipher, however, was 73.9147, which means that the encrypted text was much more dispersed and varied than the standard unencrypted English text. This means that the encrypted text is spread over a much wider range from the mean and thus indicates an improved cipher over monoalphabetic ciphers such as the Caesar Cipher or the Substitution Cipher.

The Shannon entropy of the unencrypted plaintext is 4.5158, which is typical of standard English writing.

The Shannon entropy of the encrypted text using the CSS cipher is 7.9836, which shows that there is more uncertainty and surprise with the encrypted text than the plain English text. There is thus more information contained in the encrypted text when using the CSS cipher, meaning that the encryption indeed makes the text contain more randomness/information and thus makes it more difficult to decrypt. This means that 8 bits per symbol would be required to encode the information in binary, as opposed to the mere 5 bits per symbol required to encode the original English text.

The modal and median letters are subject to change depending on whatever texts and keys are used to encrypt the text using the CSS cipher, and so they are thus meaningless besides the fact that the modal and median letters vary wildly and most definitely almost always change after encryption of text (barring extremely unlikely coincidences.)

The index of coincidence (without normalization) of the unencrypted plaintext is 0.06609, which is very close to the expected IOC of normal English text of 0.067 [1]. The index of coincidence of the text that has been encrypted by the CSS cipher method, however, is 0.00392, which is very far away from the expected ICc of normal English text. This means that the newly encrypted text has a very different frequency distribution of characters when compared to the uniform distribution, meaning that the cipher is more difficult to crack than something like the substitution cipher, which maintains the same IOC after encryption.

Summary: Encryption and decryption of binary files was successfully achieved using the Content Scrambling System cipher. The files were then analyzed by several different data statistics, including Shannon entropy, standard deviation, mean, mode, median, and index of coincidence.

Appendix

Fox_Project_2_CSS.py:

```
# Aaron Fox
# CECS 564-01
# Spring 2020
# Dr Desoky
# This file encrypts and decrypts binary files based on the Content Scrambling System cipher
# originally devised in 1996. The encryption uses two linear-feedback shift registers (LFSR) to generate
# pseudorandom numbers based on a seeded 40-bit key along with a full adder to add the result
# of the pseudorandom numbers generated from those LFSRs. The result of the full adder is then
# XORd bitwise with the input characters (padding is added to make the inputs of size 8-bits as needed.)
# The result of XORing each character of the text to encrypt and the full adder result is then
# converted to its decimal equivalent which is then converted to its equivalent Unicode value
# that is then written to an output binary file.

# For graphing frequency distributions
import matplotlib.pyplot as plt

# For log in Shannon Entropy and sqrt for standard deviation
import math

# get_index_of_coincidence returns the index of coincidence (IOC) of a text
def get_index_of_coincidence(text):
    # Calculate frequency of each char
    char_frequency = {}
    for i in range(256):
        char_frequency[chr(i)] = 0

    for char in text:
        char_frequency[char] = char_frequency[char] + 1

    sum = 0
    N = len(text)
    for key in char_frequency.keys():
        sum = sum + (char_frequency[key] * (char_frequency[key] - 1))

    return sum / (N * (N - 1))

# For getting the mean of text
def get_mean_of_text(text):
```

```
    sum = 0
    count = 0
    for char in text:
        sum = sum + ord(char)
        count = count + 1

    return sum / count

# For calculating Shannon Entropy of text
def get_shannon_entropy(text):
    count = 0
    dict = {}
    for char in text:
        count = count + 1
        if char not in dict:
            dict[char] = 1
        else:
            dict[char] = dict[char] + 1

    # get probability of each letter occurring of each letter
    for key in dict.keys():
        dict[key] = dict[key] / count

    # sort dictionary
    dict = {k: v for k, v in sorted(dict.items(), key=lambda item: item[0])}

    running_sum = 0
    for key in dict.keys():
        running_sum = running_sum + dict[key] * math.log(dict[key], 2)

    return -1 * running_sum


# For obtaining standard deviation of text
def get_stand_dev_of_text(text):

    mean = get_mean_of_text(text)
    running_sum = 0
    count = 0
    for char in text:
        running_sum = running_sum + (ord(char) - mean) ** 2
        count = count + 1
    return math.sqrt(running_sum / count)

# For graphing the character distribution
def graph_char_distribution(text):
    dict = {}
    for char in text:
        if char not in dict:
            dict[char] = 1
        else:
            dict[char] = dict[char] + 1

    # sort dictionary
    dict = {k: v for k, v in sorted(dict.items(), key=lambda item: item[0])}

    plt.bar(dict.keys(), dict.values())
    plt.xlabel('Character')
    plt.ylabel('Frequency')
    plt.title('Frequency Distribution of Characters')
    plt.show()

# Prototype Linear-Feedback Shift Register method that will
# be replaced by the generator method below it
# INPUT: taps: The equivalent primitive polynomial used (e.g. for polynomial x^15 + x + 1, taps=[15, 1])
#        seed: Seed of LSFR to be used in binary, e.g. 40 bit key of '0001100111100110000000110100000000001100'
```

```
# OUTPUT: None (just prints)
def LFSR(taps, seed):
    s = seed
    xor_output = 0

    init_pass = 0
    cycle_length = 0
    print(s)
    while (s != seed or init_pass == 0):# and cycle_length < 5:
        cycle_length = cycle_length + 1
        init_pass = 1
        for tap in taps:
            xor_output = xor_output + int(s[len(s)-tap])
            # xor_output = xor_output + int(s[tap-1])

        if xor_output % 2 == 0.0:
            xor_output = 0
        else:
            xor_output = 1
        s = str(xor_output) + s[0:len(s) - 1]
        xor_output = 0
        print(s)
    # Print out final seed also to show cycle
    print(s)
    print("Cycle length: " + str(cycle_length))


# Generator function so state of local variables is saved and
# yields the next shifted bit to be used by the full adder without repitive iterations
# INPUT: taps (list of ints): The equivalent primitive polynomial used (e.g. for polynomial x^15 + x + 1, taps=[15, 1])
#        seed (string): Seed of LSFR to be used in binary, e.g. 40 bit key of '0001100111100110000000110100000000001100'
# OUTPUT: The next shifted bit of the generator, taking account of the current place and state of all local variables
def LFSR_generator(taps, seed):
    s = seed
    xor_output = 0
    yield s[len(s)-1]
    while 1:
        for tap in taps:
            xor_output = xor_output + int(s[len(s)-tap])
        if xor_output % 2 == 0.0:
            xor_output = 0
        else:
            xor_output = 1
        s = str(xor_output) + s[0:len(s) - 1]
        xor_output = 0
        yield s[len(s)-1]

# This code shows how each bit can be generated in bytes (8 bits at a time)
# test_gen = LFSR_generator((4, 1), '0001')
# for i in range(48):
#     if i % 8 == 0:
#         print()
#     print(next(test_gen), end='')

# Converts a string of binary letters to an integer in decimal
# INPUT: Takes in a text string of binary numbers
# OUTPUT: Outputs an integer decimal number of the equivalent decimal of the input binary string
def binary_to_decimal(binary):
    decimal = 0
    for i in range(len(binary)):
        decimal = decimal + int(binary[len(binary) - 1  - i]) * 2 ** i
    return decimal

# Code for 8 bit full adder STARTS here

# Half adder implementation in Python to be used in full adder
```

```
# INPUT: Two bits to be half-added
# OUTPUT: Tuple (sum, carry bit)
# NOTE: ^ is bitwise XOR operator
def half_adder(bit_1, bit_2):
    return (bit_1 ^ bit_2, bit_1 and bit_2)

# Full adder implementation in Python using half adders
# INPUT: two bits to be full-added and an optional carry bit which defaults to 0
# OUTPUT: Tuple (sum, carry bit)
def full_adder(bit_1, bit_2, carry_bit=0):
    sum_1, carry_1 = half_adder(bit_1, bit_2)
    sum_2, carry_2 = half_adder(sum_1, carry_bit)

    return (sum_2, carry_1 or carry_2)

# Adds to bits of the same length together, rolling over if needed
# This is named 'x_bit_full_adder' because it works for bits of any x length
# INPUT: bits_1 (and bits_2): string of binary bits
# OUTPUT: Resulting binary string of the result of the X bit full adder
def x_bit_full_adder(bits_1, bits_2):
    # Initial carry bit is 0 per CSS Project 2 prompt
    carry_bit = 0
    # Store results into string backwards so that it can be reversed later
    result = ''
    for i in range(len(bits_1) - 1, -1, -1):
        sum_bit, carry_bit = full_adder(int(bits_1[i]), int(bits_2[i]), carry_bit)
        result = result + str(sum_bit)
    # Return result but inverse
    return result[::-1]

# Code for 8 bit full adder ENDS here

# convert_bytes_to_binary_number onverts bytes to binary numbers and adds padding to numbers if needed
# INPUT: list of bytes to convert to binary, e.g. list of bytes of [25, 230, 3, 64, 12]
#        converts to 00011001111001100000001101000000000001100 in binary
# OUTPUT: a 40 bit binary string
def convert_bytes_to_binary_number(bytes):
    binary_40_bit_string = ''
    for byte in bytes:
        binary = str(bin(byte))[2:]
        # Add padding of 0's to binary if needed
        binary = '0' * (8- len(binary)) + binary
        binary_40_bit_string = binary_40_bit_string + binary
    print("converted 40-bit key is " + binary_40_bit_string)
    return binary_40_bit_string


# xor_binary takes in two binary strings, adds padding to make them bytes
# and then XORs them and returns the result as a string
# INPUT: binary_1 (and binary_2): strings of binary bits
# OUTPUT: result: new binary string of XORing the original binary strings
def xor_binary(binary_1, binary_2):
    # Make sure both binary strings are of length 8 by padding them with 0s at beginning if necessary
    binary_1 = '0' * (8 - len(binary_1)) + binary_1
    binary_2 = '0' * (8 - len(binary_2)) + binary_2

    result = ''
    for i in range(len(binary_1)):
        result = result + str((int(binary_1[i]) ^ int(binary_2[i])))

    return result

# encrypt_css encrypts a given text using a key and 2 LFSRs per the Content Scrambling System
# technique described in the question prompt
# INPUT: key: 40 bits long, or 5 bytes, each byte 0-255, separated by commas in a list e.g. [243, 22, 49, 105, 6]
#        text_to_encrypt: a string of text to be encrypted
```

```
# OUTPUT: (None) The encrypted text is written to a binary file for decrypting later
def encrypt_css(key, text_to_encrypt):
    binary_40_bit_key = convert_bytes_to_binary_number(key)
    # First 2 bytes (first 16 bits) are to be used in first LSFR
    intialization_key_for_LSFR_1 = binary_40_bit_key[0:16]
    # Append 1 to end of intitialization key to ensure key is not all 0's
    # Therefore LSFR 1 is 17 bits long, as CSS prompt specifies
    intialization_key_for_LSFR_1 = intialization_key_for_LSFR_1 + '1'


    # Last 3 bytes (last 24 bits) are to be used in second LSFR
    intialization_key_for_LSFR_2 = binary_40_bit_key[16:40]
    # Append 1 to end of intitialization key to ensure key is not all 0's
    # Therefore LSFR 2 is 25 bits long, as CSS prompt specifies
    intialization_key_for_LSFR_2 = intialization_key_for_LSFR_2 + '1'

    LFSR_1_generator = LFSR_generator((15, 1), intialization_key_for_LSFR_1)
    LFSR_2_generator = LFSR_generator((15, 5, 4, 1), intialization_key_for_LSFR_2)

    # Printing out text
    if len(text_to_encrypt) > 25:
        print("text_to_encrypt == " + text_to_encrypt[0:20] + "...")
    else:
        print("text_to_encrypt == " + text_to_encrypt)

    # File to encrypt text to
    output_file_name = r"C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project 2\encrypted_CSS_text.txt"
    encrypted_file = open(output_file_name, "w", encoding="latin1")
    # Initial carry bit of full adder is 0
    # For every character in text, get next shifted 8 bits from both registers
    for char in text_to_encrypt:# LEFT OFF HERE
        # Get 8 binary bits
        register_1_binary = ''
        register_2_binary = ''
        for i in range(8):
            register_1_binary = register_1_binary + str(next(LFSR_1_generator))
            register_2_binary = register_2_binary + str(next(LFSR_2_generator))


        full_adder_result = x_bit_full_adder(register_1_binary, register_2_binary)
        # Encryption and decryption are done by bitxor of input bytes with the keystream bytes
        # print("str(bin(ord(char))[2:] == " + str(bin(ord(char))[2:]))
        result = xor_binary(full_adder_result, str(bin(ord(char))[2:]))
        # print("result == " + result)
        # Write encrypted text to file
        encrypted_file.write(chr(binary_to_decimal(result)))
    print("Encrypted file text is stored in " + output_file_name)


    encrypted_file.close()

# decrypt_css decrypts a given text using a key and 2 LFSRs per the Content Scrambling System
# technique described in the question prompt
# INPUT: key: 40 bits long, or 5 bytes, each byte 0-255, separated by commas in a list e.g. [243, 22, 49, 105, 6]
#        text_to_decrypt: a string of text to be decrypted
# OUTPUT: (None) The decrypted text is written to a binary file
def decrypt_css(key, text_to_decrypt):
    binary_40_bit_key = convert_bytes_to_binary_number(key)
    # First 2 bytes (first 16 bits) are to be used in first LSFR
    intialization_key_for_LSFR_1 = binary_40_bit_key[0:16]
    # Append 1 to end of intitialization key to ensure key is not all 0's
    # Therefore LSFR 1 is 17 bits long, as CSS prompt specifies
    intialization_key_for_LSFR_1 = intialization_key_for_LSFR_1 + '1'


    # Last 3 bytes (last 24 bits) are to be used in second LSFR
```

```
    intialization_key_for_LSFR_2 = binary_40_bit_key[16:40]
    # Append 1 to end of intitialization key to ensure key is not all 0's
    # Therefore LSFR 2 is 25 bits long, as CSS prompt specifies
    intialization_key_for_LSFR_2 = intialization_key_for_LSFR_2 + '1'

    LFSR_1_generator = LFSR_generator((15, 1), intialization_key_for_LSFR_1)
    LFSR_2_generator = LFSR_generator((15, 5, 4, 1), intialization_key_for_LSFR_2)
    if len(text_to_decrypt) > 25:
        print("text_to_decrypt == " + text_to_decrypt[0:20] + "...")
    else:
        print("text_to_decrypt == " + text_to_decrypt)
    # File to decrypt text to
    output_file_name = r"C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project 2\decrypted_CSS_text.txt"
    decrypted_file = open(output_file_name, "w", encoding="latin1")
    # Initial carry bit of full adder is 0
    # For every character in text, get next shifted 8 bits from both registers
    for char in text_to_decrypt:
        # Get 8 binary bits
        register_1_binary = ''
        register_2_binary = ''
        for i in range(8):
            register_1_binary = register_1_binary + str(next(LFSR_1_generator))
            register_2_binary = register_2_binary + str(next(LFSR_2_generator))

        full_adder_result = x_bit_full_adder(register_1_binary, register_2_binary)
        # Encryption and decryption are done by bitxor of input bytes with the keystream bytes
        # print("str(bin(ord(char))[2:] == " + str(bin(ord(char))[2:]))
        result = xor_binary(full_adder_result, str(bin(ord(char))[2:]))
        # print("result == " + result)
        # Write encrypted text to file
        decrypted_file.write(chr(binary_to_decimal(result)))
    print("Decrypted file text is stored in " + output_file_name)

    decrypted_file.close()


# Encrypting text
text_file_path = r'C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project 2\helloworld.txt'
text_file_path = r'C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project 2\The_Lottery_Shirley_Jackson.txt'
# text_file_path = r'C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project 2\Darwin.txt'

file_to_encrypt = open(text_file_path, 'r', encoding='latin1')
text_to_encrypt = file_to_encrypt.read()
# graph_char_distribution(text_to_encrypt)
print('unencrypted text mean == ' + str(get_mean_of_text(text_to_encrypt)))
print('unencrypted text std dev == ' + str(get_stand_dev_of_text(text_to_encrypt)))
print("unencrypted text shannon entropy == " + str(get_shannon_entropy(text_to_encrypt)))
print("IOC of unencrypted text == " + str(get_index_of_coincidence(text_to_encrypt)))


file_to_encrypt.close()
encrypt_css([243, 22, 49, 105, 6], text_to_encrypt)

# Decrypting text
# To decrypt, simply encrypt again since the key is always involutory since
# XOR(XOR(key, text), key) = text per the rules of XOR
encrypted_text_file_path = r"C:\Users\aaron\Classes_11th_Semester\CECS 564\CECS-564-Cryptography\Project 2\encrypted_CSS_text.txt"
encrypted_text_file = open(encrypted_text_file_path, "r", encoding="latin1")
encrypted_string = encrypted_text_file.read()
# graph_char_distribution(encrypted_string)
print('encrypted text mean == ' + str(get_mean_of_text(encrypted_string)))
print('encrypted text std dev == ' + str(get_stand_dev_of_text(encrypted_string)))
print("encrypted text shannon entropy == " + str(get_shannon_entropy(encrypted_string)))
print("IOC of encrypted text == " + str(get_index_of_coincidence(encrypted_string)))
encrypted_text_file.close()
decrypt_css([243, 22, 49, 105, 6], encrypted_string)
```

```
# R1 LSFR of CSS
# LFSR((15, 1), '00000000000001000')

# R2 LSFR of CSS
# LFSR((15, 5, 4, 1), '000000000000000000001000')

# LFSR((4, 1), '0001')
# LFSR((4, 3), '0110')
# LFSR((3, 2), '011')
# List of primitive polynomials: http://users.ece.cmu.edu/~koopman/lfsr/index.html
# e.g. for Text file of size of 4 bits, 9 = 1001 = x^4 + x + 1 and C = 1100 = x^4 + x^3 + 1
```

References

[1] "Crypto-IT," *Index of Coincidence | Cryptography*. [Online]. Available: http://www.crypto-it.net/eng/theory/index-of-coincidence.html. [Accessed: 23-Mar-2020].