SPECIFICATION FOR CAESAR CYPHER / CYRPTANALYSIS TOOL

Input file: plaintextfile.txt

Output file: ciphertextfile.txt

(files must be present in .sln root directory)

Input file can be any characters present on a standard 103 key US keyboard. Program will apply Caesar encryption to only characters a-z, leaving all whitespace, spec char, etc. alone.

Output file will be all upper case with maintained whitespace, special characters, etc.

The key must be an integer and be provided by the user at runtime.

Program will initially prompt user with the option to Encrypt or Decrypt.

Once selected, the use will be prompted for a key.

If Encrypt is selected, the program will encrypt the contents of the input file via Caesar using the provided key and output the results to the output file.

If Decrypt is selected, the program will decrypt the contents of the output file via Caesar using the provided key and output the results to the output file.

Part II:

For our Caesar Cryptanalysis, the program will prompt you to either enter a string of characters, or input a .txt file.

Once selected, the program will decrypt the cipher file (or string) using every possible key.

These outputs will be compared to a bank of the most common words and will attempt to select the decryption that matches the most number of keywords.

If there is a match, the program will output the “correct” decryption, otherwise it will notify the user it is unable to determine the correct plain text.

PART V:

Prove that bitwise XOR can be used to encrypt and decrypt strings using text and a key.

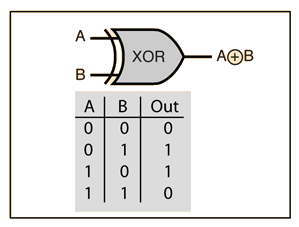
Computers today use a system to store character information as binary values. Let’s use the ASCII system of encoding, where each character producible by a standard keyboard can be represented as a unique 2-digit HEX value. In other words, we can say there is a function F(x) such that each character is bijectivity mapped to a unique HEX value. Each hex value represents a unique character (see [more](https://en.wikipedia.org/wiki/ASCII)).

In this way, any character (and by extension string) can be represented as a unique binary value.

Similarly, a “key” of type integer can be represented as a binary value, via conversion from base 10 to base 2.

Encryption systems work by allowing the key to modify the original string in a way that reversing the process yields the original result. For our example, we have two binary values, the “plaintext” binary value and the “key” binary value.

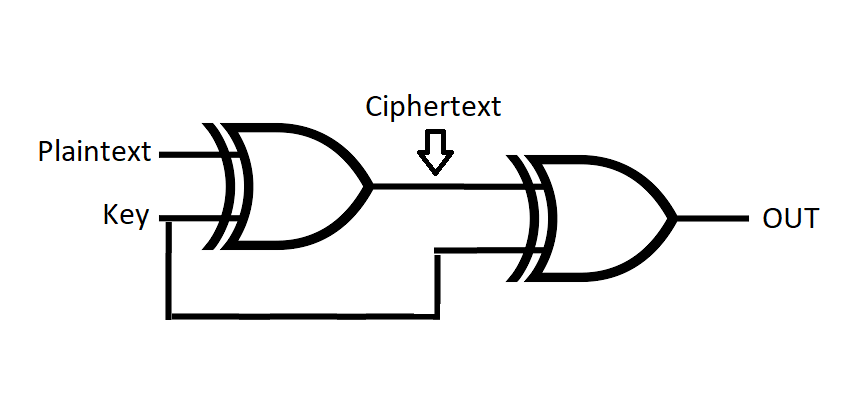
Using our knowledge of logic gates, we know that the XOR gate provides a high output when the inputs are not the same, as in the table below:



This may look familiar to a freshman student of computer science, as this is the logic gate used to add (mod 2) binary numbers together.

Since addition (by definition) is a bijection from the field of integers to integers, it is reflexive and can be “undone” using the additive inverse.

For binary, the additive inverse function is the additive function (XOR). Given A XOR B = C, C XOR B = A and C XOR A = B.

This is illustrated by tying two XOR gates together, like below:

|  |  |  |  |
| --- | --- | --- | --- |
| Plaintext | Key | Ciphertext | OUT |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |

It is shown in the above Truth Table that plaintext is equivalent to OUT. Therefore, an XOR function output on our plaintext binary number with the key binary number will be equivalent to the output XORed with the key.

PART VI:

Drew: 40%

Caston: 60%