Cryptology (2)

Classic Ciphers and Modular Arithmetic

(less math version)

John York, Blue Ridge Community College

http://www.brcc.edu

Much of the information in this course came from “Understanding Cryptography” by Christoff Parr and Jan Pelzl, Springer-Verlag 2010
Much of the classical cryptography material and most Python scripts came from “Cracking Codes with Python” by Al Sweigart, NoStarch Press 2018
Also helpful, “Cryptography Engineering” by Ferguson, Schneier, and Kohno, Wiley Publishing, 2010

Classic Ciphers

- Not in common use, except as Capture The Flag (CTF) problems
- Substitution (or Replacement) Cipher
  - Each letter of the alphabet is replaced by another
  - A -> M, B -> Q, etc.
  - Caesar cipher is a simple example
  - Susceptible to frequency analysis
    - Most common letters in English are E, then T, then I, …
- Transposition (or Permutation) Cipher
  - Jumble the order of the plaintext letters
  - Susceptible to brute force attacks

Note: The classic ciphers discussed here are no longer considered to be encryption since they are easily cracked by modern computers. The main reason we discuss them is that they are a good way to introduce the modular arithmetic we will use in later lessons on modern encryption.

However, modern symmetric encryption uses both substitution and transposition in clever ways to provide secure encryption.

Caesar (or Shift) Cipher

- Symbols shifted by a fixed number (three in the example below)

Letter	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
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Shifted	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

- Caesar in Python

(From Cracking Codes with Python by Sweigart)

```
SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
length = len(SYMBOLS)
shift = 3
plaintext = 'CAESAR'
ciphertext = ''
for letter in plaintext:
    index = SYMBOLS.find(letter)
    ciphertext += SYMBOLS[(index + shift) % length]
print(ciphertext)
```

SYMBOLS is just a string of the letters we will use.

SYMBOLS[i] gives us the i^{th} letter in SYMBOLS. SYMBOLS[4] is E, SYMBOLS[0] is A

SYMBOLS.find(letter) finds the index number of a letter.
SYMBOLS.find('A') is 0, SYMBOLS.find('E') is 4

In Python, the "%" sign is the modulo operator.
30 % 26 is 30 mod 26 = 4
5 % 26 is 5 mod 26 = 5
-4 % 26 is -4 mod 26 = 22

Modular Addition

- Key part of Python Caesar script is:
  - ciphertext += SYMBOLS[(index + shift) % length]
- Modular addition "wraps around"
  - Index of 'Z' is 25, we have 26 symbols (0-25)
  - 'Z' shifted by 3 is index 28, which doesn't exist
  - Index "wraps" by starting over at 0
    - …, 25, 0, 1, 2, 3, …
  - 'Z' shifted by 3 is index 2, or 'C'
- Math: cipherIndex = (index + shift) mod length

In our case, with 26 letters (symbols), length is 26 and we are using modulo 26.
If our shift is 3, we are shifting 3 to the right.

For A, SYMBOLS.find('A') = 0, so the index is 0.
cipherindex = (0 + 3) mod 26 = 3 mod 26 = 3
SYMBOLS[3] is D
+= means that D is added to the end of string ciphertext
A becomes D when encrypted

For Z, SYMBOLS.find('Z') = 25, so the index is 25.
cipherindex = (25 + 3) mod 26 = 28 mod 26 = 2
SYMBOLS[2] is C
+= means that C is added to the end of string ciphertext
Z becomes C when encrypted

Decrypt Caesar Cipher

- Encrypt: cipherIndex = (index + shift) mod length
- Decrypt: index = (cipherIndex – shift) mod length
- In our example shift = 3 and length = 26
  - Encrypt: cipherIndex = (index + 3) mod26
  - Decrypt: index = (cipherIndex – 3) mod26
- Decrypt code same as encrypt, just change shift
- If shift is 13, encrypt and decrypt are the same
- Multiple encryptions add no security
  - Caesar shift 3 followed by Caesar shift 4 is just Caesar shift 7

Assuming the shift was 3 for these steps.

If ciphertext is D, SYMBOLS.find('D') is 3, so cipherIndex is 3.
Index = (cipherIndex -3) % 26 = (3 - 3) % 26 = 0
SYMBOLS[0] is A, so plaintext is A

If ciphertext is C, SYMBOLS.find('C') is 2, so cipherIndex is 2.
Index = (cipherIndex -3) % 26 = (2 - 3) % 26 = -1 % 26 = 25
SYMBOLS[25] is Z, so plaintext is Z

ROT13 (rotate 13) is a simple Caesar cipher that was in widespread use in the early days of the Internet. It was mostly used to prevent spoilers; if you didn't want the reader to see an answer inadvertently, you would encode the text with ROT13. If the reader wanted to see the answer, they would just apply ROT13 again to decode it.

Affine Cipher, a more general case

- Instead of simple shift (add to index), Affine cipher also multiplies
  - cipherIndex = A * index + B
- Jumbles symbols rather than shifting
  - Symbols[index] ABCDEFGHIJKLMNOPQRSTUVWXYZ
  - Symbols[3 * index] ADGJMPSVYBEHKNQTWZCFILORUX
  - length = len(SYMBOLS) = 26 shift B = 0 in this case
- Problem: A and length cannot have a common factor (A and length must be relatively prime)
  - Symbols[13 * index] ANANANANANANANANANANANANAN
  - Symbols[8 * index] AIQYGOWEMUCKSAIQYGOWEMUCKS
- This is one reason why prime numbers are common in encryption

Symbols[13 * index]
index = 1 then (13 * index) mod 26 = (13) mod 26 = 13
index = 2 then (13 * index) mod 26 = (26) mod 26 = 0
index = 3 then (13 * index) mod 26 = (39) mod 26 = 13
index = 4 then (13 * index) mod 26 = (52) mod 26 = 0
The only characters that appear in the ciphertext are SYMBOLS[0] = A and SYMBOLS[13] = N

Symbols[8 * index]
index = 1 then (8 * index) mod 26 = (8) mod 26 = 8
index = 2 then (8 * index) mod 26 = (16) mod 26 = 16
index = 3 then (8 * index) mod 26 = (24) mod 26 = 24
index = 4 then (8 * index) mod 26 = (32) mod 26 = 6
index = 5 then (8 * index) mod 26 = (40) mod 26 = 14
<skip>
index = 14 then (8 * index) mod 26 = (112) mod 26 = 8
index = 15 then (8 * index) mod 26 = (120) mod 26 = 16
index = 16 then (8 * index) mod 26 = (128) mod 26 = 24
index = 17 then (8 * index) mod 26 = (134) mod 26 = 6

index = 18 then (8 * index) mod 26 = (144) mod 26 = 14
index 13 through 25 is just a repetition of 0 through 12

GCD and Relatively Prime Numbers

- Greatest Common Divisor (GCD) of two numbers is the largest number that can divide into both, with no remainder.
  - GCD(36, 45) = 9
- GCD(a, b) = 1 means the numbers are relatively prime.
  - GCD(44, 45) = 1
- Euclid's Algorithm computes GCD quickly.
  - Divide larger number by smaller and take the remainder b % a
  - Continue while the smaller number is > 0
  - What's left is GCD

```
def gcd(a, b):
    # Return the Greatest Common Divisor of a and b using Euclid's Algorithm
    while a != 0:
        a, b = b % a, a
    return b
```

https://inventwithpython.com/cracking/

gcd(36, 45) a = 36, b = 45
a = b % a = 45 % 36 = 9 b = a = 36
a = 9, b = 36

a= b%a = 36 % 9 = 0 b = a = 9
a = 0, b = 9
a = 0 so quit, answer is 9
The GCD of 36 and 45 is 9. 9 * 4 = 36, 9 * 5 = 45

gcd(44, 45) a = 44, b = 45
b = b % a = 45 % 44 = 1 b = a = 44
a = 1, b = 44

a = b%a = 44 % 1 = 0 b = a = 1
a = 0, b = 1
a = 0 so quit, answer is 1
Therefore, 44 and 45 are relatively prime. They have no factors in common other than 1.

Affine Decryption and the Modular Inverse

- Affine Encrypt: cipherIndex = (A * index + B) mod length
- Affine Decrypt: index = (cipherIndex – B) / A ????
  - Division does not work in modular arithmetic
  - In modular arithmetic, we multiply by the inverse
  - $A^{-1} * A = 1 \bmod length$
  - If A is 3 and length is 26, $9 * 3 \bmod 26 = 27 \bmod 26 = 1$
  - 3 and 9 are multiplicative inverses in modulo 26
  - Index = $(cipherIndex - B) * A^{-1} \bmod length$
- For $A^{-1} \bmod length$ to exist, A and length must be relatively prime

There is no division in modular arithmetic. If our set is the Integers [0 .. 25], what do we do with 3/2? Our set does not include 1.5.

Instead, define the modular inverse for multiplication. If X * Y mod length = 1, then X and Y are inverses. 3 * 9 % 26 = 1, so 3 and 9 are inverses in modulo 26. However, the modular inverse of a number may not always exist--think of the problem with Affine encryption two slides ago where the multiplier A and the length contained a common factor. A = 2 and length = 26 only gave answers of A and N

A and length must be relatively prime. This means that GCD(A, length) = 1

If A and length have a common factor, $A^{-1} \bmod length$ does not exist, and that value of A cannot be used for Affine encryption.

Key Point—Multiplicative Inverse

- When Multiplicative Inverse does not exist, multiplication does not give a unique answer—remember Affine cipher, mod 26
  - Symbols[index] ABCDEFGHIJKLMNOPQRSTUVWXYZ
  - Symbols[13 * index] ANANANANANANANANANANANANAN
  - Symbols[8 * index] AIQYGOWEMUCKSAIQYGOWEMUCKS
- Some encryption develops ways to work around this
  - AES defines new operations to replace multiplication and addition
  - Diffie-Hellman excludes numbers that do not have inverse
- Some encryption makes use of this property
  - RSA is based on a modulus that is not prime

Computing Modular Multiplicative Inverses

- Brute force →
  - Try every number until a*i mod(n) = 1
    (Should also check gcd(a, n) == 1 before start to ensure inverse exists)
- Extended Euclidean Algorithm
  - Compact, but not simple

```
>>> a = 11
>>> n = 26
>>> for i in range(n):
        if a * i % n == 1:
            break
>>> i
19
>>> 19*11 % 26
1
```

The code in the top right is simple Python that incrementally tests numbers to see if they are multiplicative inverses. The Python range(n) function returns a list of integers between 0 and n-1. In our case n = 26, so it gives a list 0, 1, 2, …, 25. The break statement causes the for loop to quit when it finds the inverse we seek, where a * i % n = 1

The findModInverse function comes from "Cracking Codes with Python." It uses an interesting feature of Python called multiple assignment. The line
u1, u2, u3 = 1, 0, a
is the same as
u1 = 1
u2 = 0
u3 = a

Wikipedia has a nice explanation and a table showing several steps in the algorithm, as well as a mathematical proof.
https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm.

However, they didn't give me any better feel for how the process works. Instead, this

lecture by Christoff Paar was great! https://www.youtube.com/watch?v=fq6SXByItUI

Modular Arithmetic in Python

- Length (number of symbols) is called modulus
- 97 mod 26 is remainder when 97 is divided by 26
- Python operators
  - 97 mod 26, or 97 % 26 will return 19
    - Wraps 3 times, winds up on 19
  - // is the integer division operator, 97 // 26 will return 3
    - 97 / 26 with no remainder—number of times it wraps
  - 3 * 26 + 19 = 97 (quotient * modulus + remainder gives us the initial number)