Cryptography for those that haven't had MATH225

Jesse Simpson (May 6, 2016)

Contents

Chapter 1

Classical Ciphers

1.1 First Cipher: Caesar Cipher

Have you heard of Julius Caesar? He invented (or maybe his general?) one of the earliest ciphers – The**Caesar cipher**.

The Caesar cipher is one of the simplest forms of encryption. It is a *cipher* where each letter in the *plaintext* is altered by a shift forward of three letters. In the examples, we ignore all non alphanumeric characters when the *ciphertext* is generated.

Section 1.1 Vocabulary Terms:

Plaintext : Ordinary or readable text before it is encrypted into ciphertext or text after being decrypted.

Ciphertext : The encrypted version of plaintext.

Cipher : A secret or disguised form of writing.

Examples for Plaintext to Ciphertext:

```
Plaintext: We had eyes, but didn't recognzie Mount Tai.

Ciphertext: zhkdghbhvexwglgqwuhfrjqclhprxqwwdl
```

Shift by 3 letters forward, so W becomes z, e becomes h ... and so on

```
Plaintext: Whatever you are, be a good one.

Ciphertext after the shift: zkdwhyhubrxduhehdjrrgrqh
```

Shift by 3 letters forward so W becomes z, h becomes k ... and so on

Examples for Ciphertext to Plaintext:

```
Ciphertext: doojrrgwklqjvpxvwfrphwrdqhqg

Plaintext: allgoodthingsmustcometoanend
```

Shift by 3 letters backwards so d becomes a, o becomes l ... and so on

```
Ciphertext: zhkdyhdphulwrulrxvgxwbwrkhosrwkhuv

Plaintext: wehaveameritoriousdutytohelpothers
```

Shift by 3 letters backwards so z becomes w, h becomes e ... and so on

Code for Encrypting Plaintext (Python):

```
import string

def cleanup(cs):
    cs = cs.lower()
    xs = []
    for c in cs:
        if c in string.ascii_lowercase:
            xs.append(c)
    return ''.join(xs)


def caesar(plainText:
    cipherText = ""
    for ch in plainText:
        if ch.isalpha():
            stayInAlphabet = ord(ch) + 3
        if stayInAlphabet > ord('z'):
            stayInAlphabet -= 26
        finalLetter = chr(stayInAlphabet)
        cipherText += finalLetter
    print(cipherText)



message = input()
message = cleanup(message)

caesar(message)
```

Code for Decypting Ciphertext (Python):

```
import string

def cleanup(cs):
    cs = cs.lower()
    xs = []
    for c in cs:
        if c in string.ascii_lowercase:
            xs.append(c)
    return ''.join(xs)


def caesar(plainText):
    cipherText = ""
    for ch in plainText:
        if ch.isalpha():
            stayInAlphabet = ord(ch) - 3
        if stayInAlphabet < ord('a'):
            stayInAlphabet += 26
        finalLetter = chr(stayInAlphabet)
        cipherText += finalLetter
    print(cipherText)



message = input()
message = cleanup(message)

caesar(message)
```

1.2 Second Cipher: Shift Cipher

The **Shift Cipher** is a more flexible version of Caesar cipher. As dicussed in the previous section, Caesar cipher performs a shift foward of 3 letters for encryption and 3 letters backwards for decryption. Shift cipher offers a choice of shift by using a *Key*.

Secions 1.2 Vocabulary Terms:

Key: An input of data used to encrypt Plaintext or decrypt Ciphertext.

Keyspace: All possible values for the key. By having a larger keyspace, it becomes harder to decrypt the key.

Private Key: A key used only between the person sending the message and receiving the message.

Public Key: A key that is available to the public,which means that everyone has access to the key.

Private Key Cipher: A Cipher that is based on a private key, if the key was public, it can then be easily broken.

Example for Encypting Plaintext with the Key:

```
Plaintext: Do or do not, there is no try.
Key: 11

Shift the letters by 11 forward so D becomes o,
o becomes z ... and so on.

Ciphertext: ozzcozyzeespcptdyzecj
```

```
Plaintext: Pokemon moon will be amazing.
Key: 19

Shift the letters by 19 forward so P becomes i,
o becomes h ... and so on.

Ciphertext: ihdxfhgfhhgpbeeuxtftsbgz
```

As we can see from the examples, each letter is shifted by a set number, meaning that it would only take up to 0-25 shifts so in order for shift cipher to be broken. At the end of the section on shift cipher, it will be shown how easy it is to break Shift Cipher.

Example for Decrypting Ciphertext with the Key:

```
Ciphertext: lwpixhiwtejgedhtduaxut
Key: 15

Shift the letters by 15 backwards so l becomes w,
w becomes h ... and so on.

Plaintext: whatisthepurposeoflife
```

```
Ciphertext: wpunwvunahrlznylhazrpssavwshf
Key: 7

Shift the letters by 7 backwards so w becomes p,
p becomes i ... and so on.

Plaintext: pingpongtakesgreatskilltoplay
```

There are two methods to breaking the shift Cipher, the first method is brute force. We apply a shift of 0 up to 25 on the Cipher text and manually look at the results.

Example Brute Force Shift Cipher without the Key:

```
Ciphertext: ozqemklsddyggvlzafykugewlgsfwfv
Key: Unknown

All possible Shift Results:
Shift: 0 :ozqemklsddyggvlzafykugewlgsfwfv
Shift: 1 :nypdljkrccxffukyzexjtfdvkfreveu
Shift: 2 :mxockijqbbweetjxydwisecujeqdudt
Shift: 3 :lwnbjhipaavddsiwxcvhrdbtidpctcs
Shift: 4 :kvmaighozzuccrhvwbugqcashcobsbr
Shift: 5 :julzhfgnyytbbqguvatfpbzrgbnaraq
Shift: 6 :itkygefmxxsaapftuzseoayqfamzqzp
Shift: 7 :hsjxfdelwwrzzoestyrdnzxpezlypyo
Shift: 8 :griwecdkvvqyyndrsxqcmywodykxoxn
Shift: 9 :fqhvdbcjuupxxmcqrwpblxvncxjwnwm
Shift: 10 :epgucabittowwlbpqvoakwumbwivmvl
Shift: 11 :doftbzahssnvvkaopunzjvtlavhuluk
Shift: 12 :cnesayzgrrmuujznotmyiuskzugtktj
Shift: 13 :bmdrzxyfqqlttiymnslxhtrjytfsjsi
Shift: 14 :alcqywxeppksshxlmrkwgsqixserirh
Shift: 15 :zkbpxvwdoojrrgwklqjvfrphwrdqhqg
Shift: 16 :yjaowuvcnniqqfvjkpiueqogvqcpgpf
Shift: 17 :xiznvtubmmhppeuijohtdpnfupbofoe
Shift: 18 :whymustallgoodthingscometoanend
Shift: 19 :vgxltrszkkfnncsghmfrbnldsnzmdmc
Shift: 20 :ufwksqryjjemmbrfgleqamkcrmylclb
Shift: 21 :tevjrpqxiidllaqefkdpzljbqlxkbka
Shift: 22 :sduiqopwhhckkzpdejcoykiapkwjajz
Shift: 23 :rcthpnovggbjjyocdibnxjhzojviziy
Shift: 24 :qbsgomnuffaiixnbchamwigyniuhyhx
Shift: 25 :parfnlmteezhhwmabgzlvhfxmhtgxgw
```

As we can see the Key was 18 since a meaningful message appeared. However, that is pretty ugly to look at, which leaves the second method, Frequency Analysis.

Frequency analysis is the process at recording the occurrence of letters appearing in a given text. Since we know 'e' is the most common occuring letter in the alphabet, we can find the most occurring letter in our encrypted data and 'assume that it is e shifted to that letter. Which can speed up our results and not have

to brute force search.

CODE EXAMPLE FOR FREQUENCY ANALYSIS TODO:

Code for Brute Force Shift Cipher Decryption

```
import string

def cleanup(cs):
    cs = cs.lower()
    xs = []
    for c in cs:
        if c in string.ascii_lowercase:
            xs.append(c)
    return ''.join(xs)


def caesar(plainText, shift):
    cipherText = ""
    for ch in plainText:
        if ch.isalpha():
            stayInAlphabet = ord(ch) - shift
        if stayInAlphabet < ord('a'):
            stayInAlphabet += 26
        finalLetter = chr(stayInAlphabet)
        cipherText += finalLetter
    print(cipherText)



message = input()
message = cleanup(message)
for i in range(0, 26):
    print("Shift:", i, ":", end="")
    caesar(message, i)
```

1.3 Third Cipher: Vigin‘ere Cipher

The Vigin‘ere Cipher is a form of encrypting plaintext by using different letter shifts based on chosen key or keyword. Such as taking the keyword "Times", we would shift our first letter of plaintext by 'T' spaces (position of T in the alphabet), second letter by 'i' spaces (position of i in the alphabet) and so on. If we extend past the length of our key so the 6th letter of the plaintext, it wraps around and is shifted by 'T' spaces.

The advantage of Vigin‘ere over Shift Cipher is due to the fact the entire text is shifted by different individual letter shifts instead of a whole message shift. Viginere keyspace is much larger compared to Shift cipher keyspace.

Example of Encrypting Viginere Cipher with Key:

```
Plaintext: To be or not to be that is the question?

Keyword: Relations

So we perform a shift based on the keyword,
so T becomes k(17 spaces), o becomes s (4 spaces),
... and so on.

Ciphertext: ksmehzbblksmempogajxsejcsflzsy
```

```
Plaintext: We need more supplies fast!

Keyword: Mec

So we perform a shift based on the keyword,
so W becomes I(12 spaces), e becomes I(4 spaces),
... and so on.

Ciphertext: iipqifystqwwbtnuiureuf
```

Example of Decrypting Viginere Cipher with Key:

```
Ciphertext: tbspzghwfysfffsrkhvhrgsoasg

Keyword: Food

So we perform a backwards shift based on the keyword,
So t becomes o(5 spaces), b becomes n(14 spaces)
... and so on

Plaintext: onemusttakecareofthemselves
```

```
Ciphertext: pcasqmjxgzzwxiikgvelfyjw

Keyword: Pressure

So we perform a backwards shift based on the keyword,
So p becomes a(15 spaces), c becomes l (17 spaces)
... and so on

Plaintext: alwaysstriveforgreatness
```

Code to Encrypt Viginere Cipher:

```
import string

def cleanup(cs):
    cs = cs.lower()
    xs = []
    for c in cs:
        if c in string.ascii_lowercase:
            xs.append(c)
    return ''.join(xs)

def Vigenere_E(k, s): #
    tracker = 0
    new_string =""
    for i in range(0, len(s)):
        new_char = ord(s[i]) + string.lowercase.index(k[tracker])
        if new_char > ord('z'):
            new_char -= 26
        new_string += chr(new_char)
        tracker += 1
        tracker %= len(k)
    print(new_string)

m = input()
m = cleanup(m)
key = input()
Vigenere_E(key, m)
```

Code to Decrypt Viginere Cipher with Key

```
import string

def cleanup(cs):
    cs = cs.lower()
    xs = []
    for c in cs:
        if c in string.ascii_lowercase:
            xs.append(c)
    return ''.join(xs)

def Vigenere_E(D, s): #
    tracker = 0
    new_string =""
    for i in range(0, len(s)):
        new_char = ord(s[i]) + string.lowercase.index(k[tracker])
        if new_char < ord('a'):
            new_char += 26
        new_string += chr(new_char)
        tracker += 1
        tracker %= len(k)
    print(new_string)

m = input()
m = cleanup(m)
key = input()
Vigenere_D(key, m)
```

Kasiski Method to Breaking Viginere: The easiest way to understand the method is through an example.

Example : Plaintext: maths is short for mathematics

Key: key

Ciphertext: werrwgcwfyvrpspwerrikkxgw

Now by looking at the Ciphertext, we can see a set of repeating characters

werrwgcwfyvrpsp**werr**ikkxgw, we can see that there is 15 spaces between the repeated letters. Therefore we know that the key is most likely the gcd of 3,5 and 15, 1 isn't considered or that would be a Caesar Shift.

The keylength is most likely the GCD of the three numbers, which is 3. This means that every fourth character was encrypted by the same shift amount. We group the characters by their shift, and then perform frequency analysis in order to determine the key. This works for large sample cases where frequencies can be matched easier.

1.4 Project Gutenberg

Project Gutenberg is a free website that provides readers with free text books. Which just so happens to make it an excellent example on showing the frequency that letters, one-grams, two-grams .. n-grams for the common english language.

There are many ways to begin this process, the first step; however is a method provided by project gutenberg to use a robot harvest to get a LIST of all of their books available for download in a certain format. You will need wget in order to use this following method.

In the command line run:

wget -w 2 -m
http://www.gutenberg.org/robot/harvest?filetypes[]=txt&langs[]=en

This will begin downloading text document files that contain a list of urls from the main project gutenberg website. This information is very important, if at anypoint the download stops, refer to the wget log in order to continue where it left off.

After all of the harvest files have been downloaded, we will need to convert them into text files in order to manipulate them. The code that I used to accomplish this was very simple.

```
import os

path = os.getcwd()
filenames = os.listdir(path)

for filename in filenames:
    os.rename(filename, filename.replace("=txt", "=txt.txt").lower())
```

This will simply iterate through all the files in the directory where your harvest files are stored, and convert them into .txt files.

After this process has been accomplished, next is to combine all of the files into one file for regex parsing. This was accomplished by using the following code:

```
import fileinput
import glob

file_list = glob.glob("*.txt")

with open('result.txt', 'w') as file:
    input_lines = fileinput.input(file_list)
    file.writelines(input_lines)
```

The code is using the glob library in order to create a list containing all of the text files in our directory. It then iterates through the list, per line, writing it to our result text file.

However that is not the final step before we begin the download process. We have to scrape away all of the html code since what we previously downloaded were html links to direct downloads for the main project gutenberg site. Using the beautiful soup library, which parses html code made it very easy to fix this problem. Here was the code used for that process:

```
#!/usr/bin/env python3
from bs4 import BeautifulSoup
import fileinput

text = open('result.txt', 'r')

soup = BeautifulSoup(text)

clean = soup.get_text()

y = open('rcleanresult.txt', 'w')
x = open('rcleanresult.txt', 'a')
x.write(clean)
```

This code is taking our result.txt file that was made from the previous code and opens it as a read file. Next it uses the BeautifulSoup library to turn it into a soup manipulatable file. By using get text function, it strips all non text of the file, which maintains the structure even without html.

Now we have a file called rcleanresult.txt which contains the download links for all of the project gutenberg books of the type specified in our wget at the start.

Exercise: Using the information given so far, what approach would you use in order to download all of the books from project gutenberg, while not getting ip-banned for consecutive downloads?

Answer: The answer that our group came up with (Thanks Dr. Liow) was to use multi-threading through the project gutenberg mirror sites.

The reason this method works is due to the fact the last part of the download url's are all the same accross all of the mirror site. Therefore, by using REGEX or regular expressions, we can parse away all but the last part of the download url.

Example of download URL:

http://www.gutenberg.lib.md.us/1/2/3/7/12370/12370-8.zip

http://www.gutenberg.lib.md.us/1/2/3/7/12370/12370.zip

By parsing away up until we see /digit and stop and .zip, we get the key part of our download links. Since mirror sites no matter what the previous part of the url is, the numbering of the books is the same across all of the sites.

By using multi-threading between loops dedicated for each mirror site, we can download through our book list.

The question is what do we do once we have all of the books? Well we need to read through each book and count our n-grams. What are n-grams: they are the occurrence of certain letter/letter combinations.

Example of N-gram: th, e, the, have, lik ...

By using the following code to scan an entire directory (where are downloaded books are stored) we created 4 text files holding 1 gram, 2 gram, 3 gram, and 4 grams of our scanned text books.

```
import glob
def save_counts(counts, filename):
    import json
    with open(filename, 'w') as f:
        json.dump(counts, f)

def load_counts(filename):
    import json
    with open(filename, 'r') as f:
        return json.load(f)

def clean_up(s):
    z = ''
    for c in s:
        if c.isalpha():# or c == ' ':
            z += c.lower()
    return z

def get_from(s, i, l):
    return s[i:i+l]

def count1(s, counts):
    """ Counts the number of single characters """
    for c in 'abcdefghijklmnopqrstuvwxyz':
        if c in counts:
            counts[c] += s.count(c)
        else:
            counts[c] = s.count(c)
    return counts;


def count(s, l, counts):
    s = clean_up(s)
    for i in range(len(s)-1):
        gram = get_from(s, i, l).lower()
        if gram in counts:
            counts[gram] += 1
        else:
            counts[gram] = 1
    return counts
```

```
def top(counts, n=None):
    if not n:
        n = len(counts)
    counts.sort(reverse=True)
    for count in counts[:n]:
        print("{:8}".format(count[1]), count[0])
    print('----------------------------------------------')

def to_list(counts):
    c = []
    for k in counts.keys():
        c.append((counts[k], k))
    return c

def save_a_book(n):
    z = 'http://www.gutenberg.org/cache/epub/{}/pg{}.txt'.format(n, n)
    import urllib
    from urllib import request
    import http.cookiejar
    jar = http.cookiejar.MozillaCookieJar()
    opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(jar))
    words = opener.open(z).read().decode()
    #words = request.urlopen(z).read().decode()
    for cookie in jar:
            print(cookie.name)
    print(jar)
    with open('a.txt', 'w') as f:
        f.write(words)

def English(s):
    if 'Language: English' in s:
        return True
    if 'Language: en' in s:
        return True
    else:
        return False
```

```
def get_beginning_of_book(s):
    z = s.index('*** START OF THIS')
    s = s[z + 4:]
    z = s.index('***')
    s = s[z:]
    return s

book_directory = input('Give me the directory name: ')
books = glob.glob(book_directory + '/*')
print (books)
one_grams = {}#load_counts('1gram.txt')
two_grams = {}#load_counts('2gram.txt')
three_grams = {}#load_counts('3gram.txt')
four_grams = {}#load_counts('4gram.txt')
five_grams = {}#load_counts('5gram.txt')
for book in books:
    with open(book, 'r') as f:
        s = f.read()
        #if English(s):
        s = get_beginning_of_book(s)
        one_grams = count1(s, one_grams)
        two_grams = count(s, 2, two_grams)
        three_grams = count(s, 3, three_grams)
        four_grams = count(s, 4, four_grams)
        five_grams = count(s, 5, five_grams)

save_counts(one_grams, '1gram.txt')
save_counts(two_grams, '2gram.txt')
save_counts(three_grams, '3gram.txt')
save_counts(four_grams, '4gram.txt')
save_counts(five_grams, '5gram.txt')
```

The code will compile a list of all of the books, scan line by line, character by character, calling one gram, two gram, three gram, four grams functions to scan the text book and update the txt files which is holding all of our ngrams.

We can then print the top results or so for our different grams after analyzing all of the texts.

Here are the most common N-grams in the english language:

N-Gram	Percent
the	7.14
of	4.16
and	3.04
to	2.60
in	2.27
a	2.06
is	1.13
that	1.08
for	0.88
it	0.77
as	0.77
was	0.74
with	0.70
be	0.65
by	0.63
on	0.62
not	0.61
he	0.55
i	0.52
this	0.51
are	0.50
or	0.49
his	0.49
from	0.47
at	0.46
which	0.42
but	0.38
have	0.37
an	0.37
had	0.35
they	0.33
you	0.31
were	0.31
their	0.29
one	0.29
all	0.28
we	0.28
can	0.22
her	0.22
has	0.22
there	0.22
been	0.22

1.5 Fourth Cipher: Hill Cipher

The Hill Cipher: is a cipher based on Matrix multiplcation. Each letter is represented by a number mod 26. In order to perform encryption, each block(Matrix) is multiplied by an invertible matrix.

In order to decrypt a message encrypted by Hill cipher, each block is multiplied by the inverse of the matrix used for encryption to obtain the original message.

Section 1.4 Vocabulary Terms:

Multiplicative Inverse is a reciprocal for a number x, denoted by $1/x$ or x^{-1}, which is a number which when multiplied by x yields the multiplicative identity, 1.

Determinant is a quantity obtained by the addition of products of the elements of a square matrix according to a given rule.

Adjugate Matrix is a matrix that has taken every element of an original Matrix and replaced each value by it's cofactor.

Example: First we determine a square key matrix (2x2, 3x3...)

K or key matrix =

$$\begin{bmatrix} 2 & 4 & 5 \\ 9 & 2 & 1 \\ 3 & 17 & 7 \end{bmatrix}$$

```
plaintext = Attack at Dawn
```

In order to encrypt this plaintext, we first break the message into strings of 3. We take our first 3 characters 'Att' and create a 3 x 1 matrix, replacing each letter with a number corresponding to each letter, such as A = 0, B = 1, ... z = 25.

The result is:

$$\begin{bmatrix} 0 \\ 19 \\ 19 \end{bmatrix}$$

We now multiply the matrix by our key

$$\begin{bmatrix} 2 & 4 & 5 \\ 9 & 2 & 1 \\ 3 & 17 & 7 \end{bmatrix} \begin{bmatrix} 0 \\ 19 \\ 19 \end{bmatrix} = \begin{bmatrix} 171 \\ 57 \\ 456 \end{bmatrix} \pmod{26} = \begin{bmatrix} 15 \\ 5 \\ 14 \end{bmatrix} = 'PFO'$$

Then we would continue encrypting the rest of the 3 character strings until we have processed all of the letters.

```
encrypted plaintext = PFOGOANPGXFX
```

With our key matrix K, let d be the determinant of K. In order to find k^{-1} such that $K * K^{-1} \equiv i \pmod{26}$ where i is the identity matrix, we can use the following formula.

$K^{-1} = d^{-1} * adj(K)$

Where d * $d^{-1} = 1 \pmod{26}$ and adj(K) is the adjugate of K.

The final step is to take the encrypted plaintext, slice it into strings of 3 again, convert to integers and multiply with our Inverted Matrix to get the original plaintext.

$$K^{-1} * \begin{bmatrix} 15 \\ 5 \\ 14 \end{bmatrix} \pmod{26} = \begin{bmatrix} 0 \\ 19 \\ 19 \end{bmatrix} =' att'$$

Continue for the rest of the encrypted plaintext to obatin the original plaintext.

Chapter 2

Basic Number Theory

2.1 General Information

Number Theory is the study of the set of positive whole numbers

```
1, 2, 3, 4, 5, 6, 7, . . . ,
```

which are often called the set of natural numbers. We want to study the relationships between different sorts of numbers. People have always separated natural numbers into a variety of different types. Here are some examples:

```
Odd Numbers: {1, 3, 5, 7, 9, 11, ...}
Even Numbers:  {2, 4, 6, 8, 10, ...}
Squre Numbers: {1, 4, 9, 16, 25, 36, ...}
Cube Numbers: {1, 8, 27, 64, 125, ...}
Prime Numbers: {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, ...}
Perfect Numbers: {6, 28, 496, ...}
Fibonacci Numbers: {1, 1, 2, 3, 5, 8, 13, 21, ...}
```

Section 2.1 Vocabulary Terms:

Ring: is a set together with two binary operators + and * which has certain axioms.

Divisibility: Term used to determine wheter a number x can divide another term y without any remainder.

A **Ring** has the following axioms :

a set R with an operation called addition: for any a, b $\in$ R, there is an element a + b $\in$ R,

and another operation called multiplication: for any a, b $\in$ R, there is an element ab $\in$ R,

which satisfies the following axioms: (i) Addition is associative, i.e. (a + b) + c = a + (b + c) for all a, b, c $\in$ R.

(ii) There is an element of R, called the zero element and written 0, which has the property that a + 0 = 0 + a = a for all a $\in$ R.

(iii) Every element a $\in$ R has a negative, an element of R written a, which satisfies a + (a) = (a) + a = 0.

(iv) Addition is commutative, i.e. a + b = b + a for all a, b $\in$ R.

(v) Multiplication is associative, i.e. (ab)c = a(bc) for all a, b, c $\in$ R.

(vi) Multiplication is distributive over addition, i.e. a(b + c) = ab + ac and (a + b)c = ac + bc for all a, b, c $\in$ R.

A **Field** ring R which has the following extra properties:

(vii) R is commutative, i.e. ab = ba, a, b $\in$ R.

(viii) R has a nonzero identity element 1.

(ix) Every nonzero element of R is invertible.

Divisibility can be shown as 3 divides 15 in the sense that 3x = 15, x, 3, 15 $\in$ Z can be solved. Also written as

$$3 \mid 15$$

Properties of Divisibility: (i) $a|a$

(ii) $a|b, b|c => a|c$

(iii) $a|b => a|bx$, $\forall$ x

(iv) $a|b, a|c, => a|b + c$

(v) $a|b, a|c => a|bx + cy$ $\forall$ x,y $\in$ Z

Now that we know the axioms for a ring R and the rules of divisibility, we can look at our first Algorithm; the Euclidian Algorithm.

2.2 Euclidian Algorithm, GCD, EEA

The Euclidian Algorithm is based on the principle that the greatest common divisor of two numbers does not change even if the larger number is replaced by its difference with the smaller number.

Such as ...

```
Given a,b  Z, a > 0, b > 0 then: a = bq + r, where 0 <= r < b
```

Example: a = 10, b = 4 then:

10 = 4 * 2 + 2 where q = 2 and r = 2

Example: a = 15, b = 7 then:

15 = 7 * 2 + 1 where q = 2 and r = 1

Greatest Common Divosor/GCD is the process of determining the greatest divisor between two numbers.

Examples:

```
Given 2 integers a,b we write gcd(a,b) where a > 0, b > 0.

          gcd(10, 28) = 2
          gcd(21, 49) = 7
          gcd(21, 42) = 21
```

We can use the Euclidian Algorithm to compute GCD as well.

```
Exmaple: a = 270, b = 192 gcd(270, 192) =

          270 = 192 * 1 + 78      gcd(270, 192)=gcd(192, 78)

          192 = 78 * 2 + 36       gcd(270, 192)=gcd(192, 78)=gcd(78, 36)

          78 = 36 * 2 + 6         gcd(270, 192)..... =gcd(36, 6)

          36 = 6 * 6 + 0          gcd(270, 192)......=gcd(6,0)

          a != 0, b = 0 = gcd(6,0) = 6 = gcd(270, 192)
```

Code for Euclidian Algorithm GCD

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

We also have the **Extended Euclidian Algorithm** which is an extension to the Euclidean algorithm, which computes, besides the greatest common divisor of integers a and b, the integers x and y such that x, y, g given a, b where ax + by = g = gcd(a,b).

Example: gcd(378, 52)

```
378 = 7(52) + 14 (q = 7, r = 14)

 52 = 3(14) + 10 (q = 3, r = 10)

 14 = 1(10) + 4 (q = 1, r = 4)

 10 = 2(4) + 2 (q = 2, r = 2)

  4 = 2(2) + 0 (q = 2, r = 0)
```

Now we can work backwards to determine x and y

```
2 = 10 - 2(4)

2 = 10 - 2(14 - 10) (4 = 14 - 1(10))

2 = 3(10) - 2(14)

2 = 3(52 - 3(14)) - 2(14) (10 = 52 - 3(14)

2 = 3(52) - 11(14)

2 = 3(52) - 11(378 - 7(52)) (14 = 378 - 7(52))

2 = 81(52) - 11(378)
```

Therefore a = 81 and b = 11

EEA Code

```
def EEA(a, b):
    x = 0
    y = 1
    lastx = 1
    lasty = 0
    while a != 0:
        q = b // a
        r = b % a
        t0 = x-lastx*q
        t1 = y-lasty*q
        b = a
        a = r
        x = lastx
        y = lasty
        lastx = t0
        lasty = t1
    gcd = b
    return gcd, x, y

a = input()
b = input()
print "GCD:", EEA(a,b)
```

2.3 Congruences

Congruence mod N is when a number is congruent or equal value in a certain modular N. This is useful for determining what the value of integers are in other mod N cases. Especially since congruency is used for our proofs to be used in the future for Fermat and Euler.

Examples:

$$2 \equiv 8 \pmod 6$$

$$2 \equiv 17 \pmod 5$$

$$2 \equiv 9 \pmod 7$$

Properties of Congruence Relations :

1. $a \equiv b \pmod N$ and $a' \equiv b' \pmod N$ then $a + a' \equiv b + b' \pmod N$
2. $a \equiv b \pmod N$ and $a' \equiv b' \pmod N$ then $a * a' \equiv b * b' \pmod N$
3. $a * (b * c) \equiv (a * b) * c \pmod N$
4. $(a + b) + c \equiv a + (b + c) \pmod N$
5. $a * (b + c) \equiv ab + ac \pmod N$
6. $a * 1 \equiv a \pmod N$
7. $a * b \equiv b * a \pmod N$
8. $a + b \equiv b + a \pmod N$
9. $a + 0 \equiv a \equiv 0 + a \pmod N$

2.4 Multiplicative Inverse

Multiplicative Inverse for Mod N is useful for determining the value for such that $ax \equiv 1 \pmod{N}$ where x = a^{-1}.

We can use EEA(Extended Euclidian Algorithm) to determine the the inverse.

Example: using 3 and 7

To find the modular inverse of 3 (mod 7), we perform the EEA

$7 = 2(3) + 1$

$1 = 7 \ \ 2(3)$

Mod both sides by 7 for:

$1 \pmod{7} \equiv 7 - 2(3) \pmod{7}$

$1 \pmod{7} \equiv -2 * 3 \pmod{7}$

There, the inverse of 3 in mod 7 is $a \equiv -2 \equiv 5 \pmod{7}$

Code to Compute Inverse

```
def EEA(a, b):
    x = 0
    y = 1
    lastx = 1
    lasty = 0
    while a != 0:
        q = b // a
        r = b % a
        t0 = x-lastx*q
        t1 = y-lasty*q
        b = a
        a = r
        x = lastx
        y = lasty
        lastx = t0
        lasty = t1
    gcd = b
    return gcd, x, y

def inv(a, n):
    gcd, x, y = EEA(a, n)
    if gcd != 1:
        return None
    else:
        return x % n

a = input()
n = input()
print "Inverse:", inv(a,n)
```

2.5 Fermat's Little Theorem

Fermat's Little Theorem or FMT states that if you have p, a where p is a prime and gcd(a,p) = 1 then: $a^{p-1} \equiv 1 \pmod{p}$

Proof: Assume S is a set of all numbers from 1 to $p1$, i.e.

$S = 1, 2, 3, ..., p1.$

Assume some function $f : SSsuchthatf(x) = ax \pmod{p}$.

We must first show that, if x ∈ S, then f(x) ∈ S. We begin by assuming the opposite; that f(x) = 0. Then $ax \equiv 0 \pmod{p}$. Since gcd(a, p) = 1, we can divide by a to obtain $x \not\equiv 0 \pmod{p}$. Since $0 \notin S, x \notin S$. This means that $f(x) \not\equiv 0$ if x ∈ S.

Suppose ∃ x, y such that f(x) = f(y), i.e. $ax \equiv ay \pmod{p}$.

Since gcd(a, b) = 1, we can divide by a. We obtain: $x \equiv y \pmod{p}$

That is, f(x) is one-to-one. Therefore, applying f to each element of S must result in distinct elements for each value of S. These values must be the values of S in some order. It follows that

$1 * 2 * 3 \ldots * (p-1)$

$\equiv f(1) * f(2) * f(3)... * f(p-1)$

$\equiv (a*1)(a*2)(a*3)...(a*(p-1))$

$\equiv a^{p-1}(1*2*3...*(p-1)) \pmod{p}$

Since each j ∈ S has the property gcd(j, p) = 1, we can divide both

sides by $(p-1)!$ to obtain:

$p - 1 \equiv 1 \pmod{p}$

This will also allow us to work on Euler's Therom now that we know how Fermat's works.

2.6 Euler-Phi and Euler's Theorem

Euler's Theorem is a generalized theorem of Fermat's Little Therom.

Section 2.6 Vocabulary Terms

phi(n) is the number of integers 1 a n such that gcd(a, n) = 1. Which has the following properties.

1. $\phi(1) = 1$

2. $\phi(mn) = \phi(m) * \phi(n)$ if gcd(m,n) = 1

3. $\phi(p) = (p - 1)$ if p is prime

4. $\phi(p^k) = p^k - p^{k-1}$ if p is prime.

With the properties of phi(n), we get Euler's Theroem.

Given a,n, If the gcd(a,n) = 1, then $a^{\phi(n)} \equiv 1 \pmod{n}$

Proof: Assume S is the set of all numbers 1 $\leq$ a $\leq$ n-1, so that gcd(a,n) = 1. We can assume some funciton f: s -¿ S such that f(x) = ax $\pmod{n}$.

First we show that if x $\in$ S, then f(x) $\in$ S. For the proof, we assume the opposite that f(x) = 0. Then ax $\equiv 0 \pmod{n}$. Since we know the gcd(a,n) = 1, we divide by a to obtain x $\equiv 0 \pmod{n}$ Since $0 \notin S, x \notin S$.. This also means that f(x) $\neq 0$ if x $\in S$.

Suppose $\exists$ x,y such that f(x) = f(y), i.e $a * x \equiv a * y \pmod{n}$. Since we know the gcd(a,b) = 1, we can divide by a, thus getting:

x $\equiv y \pmod{n}$

This means that f(x) is 1 to 1. Therefore by taking f and applying to each element of S, it must result in distinct elements for each value that belongs to S. These values must also be the some order that it follows:

$$\prod_{x \in S} x$$

$$\equiv \prod_{x \in S} f(x)$$

$$\equiv a^{\phi(n)} \prod_{x \in S} \pmod{n}$$

Since we know that each j $\in$ S has the property of gcd(j,n) = 1, we can divide both sides by $\prod_{x \in S}$ to get the results:

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Chapter 3

RSA

3.1 RSA Encryption

RSA or Rivest-Shamir-Adleman is an algorithm that was created during the 1970s that is still used in modern day cryptography. RSA is a type of public key cryptosystem, which can also be referred to as asymmetric cryptography, since it uses the combination of two different keys for encryption and decryption; however, the keys are mathematically linked together.

Compared to the previous types of ciphers that we have previously learned about, RSA uses a form of public key cryptography. Meaning to say that we do not have to sneak around our key in private to the person we are trying to send the message to.

Section 3.1 Definitions:

Public-Key: A crytography system that uses two keys, a public key which is known to everyone and a private/secret key known only to the recipient of the encrypted message.

A couple issues someone might have:

1. How could two seperate people who do not meet in person, send encrypted data without first sending the key?

2. How does the person receiving the message decrypt the data, yet a 3rd party can not?

RSA has simple solutions to those issues, which is what makes it considered to be one of the most secure forms of cryptographic algorithms to date. It uses prime factorization with large primes to hide the information.

Example:

Alice has a message she wants to send to Bob. Bob will then select two large primes which are distinct. He then multiplies them together. For clarification purposes we will call them primes A, B. The equation is as follows:

$$N = A * B$$

Where N is the resulting number

Bob will now select a number to the exponent. We will call this number F. Now we have:

$$gcd(F, (A-1)(B-1)) = 1$$

Finally Bob computes a number G such that:

$$G * F \equiv 1 \pmod{(A-1)(B-1)}$$

Bob publishes the numbers N and G, while keeping A, B, and F private.

Alice will then use though publics keys and encrypt her message M.

$$C \equiv M^C \pmod{N}$$

Alice proceeds to send encrypted message C to Bob.

Bob then can simply decrypt the message by using equivalence:

$$M \equiv C^F \pmod{N}$$

3.2 RSA Proof

Here we will show the proof on why $M \equiv C^D \pmod{N}$. To prove this, we should think back to Euler's theorem.

Euler's Theorem:

If gcd(M,N) = 1, then $A^{\phi(N)} \equiv 1$

In our example, we have:

$$\phi(N) = \phi(AB) = (A-1)(B-1)$$

Now, suppose that gcd(M,N) = 1. We can assume that M most likely does not have A or B as a factor. Even though it is not for certain that this is the case, since A and B are very large primes, they are most likely not a factor of M.

We can create a number called K, we can then say

$F * G = 1 + K\phi(N)$; however, why can we say this? Because of this fact: $F * G \equiv 1 \pmod{()\phi(N)}$

Therefore:

$$C^G \equiv (M^F)^G \equiv M^{L+K\phi(N)} \equiv M(M^{\phi(N)})^K \equiv M(L^K) \equiv M \pmod{N}$$

3.3 Fermat's Primality Test

Fermat's Primality Test is a test that is based off of Fermat's Little Therem. It works like this.

Let $n > 1 \in Z$. We choose $a \in Z$ such that $1 < a < (n-1)$. If $a^{n-1} \equiv 1 \pmod{N}$, then n is composite. If $a^{n-1} \equiv 1 \pmod{N}$, then n is probably a prime.

Fermat's Primality Test can guarantee if a number is composite. However, it is important to understand that this test only probabilistically tells us if a number is prime. While the probablity is very good, it still can not guarantee with 100

Note: For this test, it is important to be able to calculate exponents as efficiently possible. The reason is due to the fact our number N is usually an extremely long number, typically hundreds of digits in length. The fastest way to do this by using a method called repeated squaring. You basically repeatedly square a value and $\pmod{N}$, then continue to do this until we have removed the largest possible power of 2. Then continue with the remaining powers.

3.4 How to Break RSA

We have talked about the impressive points behind RSA; however, can Eve break RSA?

Suppose Eve is trying to break RSA encryption. We begin by looking at the only pieces of information that Eve posses at the start.

1. The ciphertext C

2. The encryption exponent F

3. The composite N

In order to break the encryption, we need to know the number G. We will assume Eve has no secret method to factor N. If for some reason she does, then most modern encryption is useless to her. If she doesn't, then the only way for Eve to compute G would require her to know $\phi(N)$. Since knowing $\phi(N)$ is the same as knowing A and B, Eve will have an extremely hard time fidning the number G. However, if there was a way for Eve to find G, there is a good probabilityshe can factor N. She can use the mothod for universal exponents.

Universal exponent: If there exists a universal exponent B such that $a^B \equiv 1 \pmod{N}$ for all A with gcd(A,N) = 1, then you can probably factor N. Since $G * F - 1$ is a multiple of $\phi(N)$, whenever gcd(A,N) = 1, we have

$$A^{D*E-1} \equiv (A^{\phi(N)})^K \equiv 1 \pmod{N}$$

3.5 Brute Force Prime Factorization

Lets quickly look and try to factor a number the 'old fashioned way' in order to demonstrate how slow this method truly is. For the exmaple, we will use the number 1631.

We know that we will only need to try numbers up to 40, since the square root of 1631 is a decimal number between 40 and 41. We can also exclude the number 1, which leaves us with the remaing factors

$$2, 3, 4, 5, \ldots 40$$

We can then reduce this list by removing all even numbers, since we know the number that we are trying to factor is an odd number. Now we are down to 19 possibilities:

$$3, 5, 7, 9, 11, \ldots 39$$

We can then reduce the list another time by removing all numbers that have prime factors themselves. Now we have 11 numbers:

$$3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37$$

Now we try all of the remaining numbers until one of them divides into the original number 1631.

$$1631/3 = 543.66$$

$$1631/5 = 326.2$$

$$1631/7 = 233$$

Since we have found the factor 7, we can see the other factor is 233. As you can see this is easy to do with small numbers; however, when related to RSA when the numbers are very large, this method is too slow.

Chapter 4

Groups

4.1 Groups

A **Group** is an algebraic structure consisting of a set of elements equipped with an operation that combines any two elements to form a third element. Such as :

$$(G, *, e)$$

Section 3.1 Vocabulary Terms:

Abelian Group: if (G, *, e)is a grup and for all $x, y \in G$ we can say

$$x * y = y * x$$

Subgroup: Given a group G under a binary operation $*$, a subset H $\subset$ G is called a subgroup of G if H also forms a group under the operation $*$.

Where G is a set, * : G x G $\rightarrow$ G is a binary operator and e $\in$ G such that is has the following properties.

1. *Closure*: For all x,y $\in$ G,$x * y \in$ G.

2. *Neutral*: For all x $\in$ G, $e * x = x = x * 3$.

3. *Inverse*: For all $x \in G$, $\exists$ y s.t $x * y = e = y * x$.

4. *Associativity*: For all $x, y, z \in G, (x * y) * z = x * (y * z)$.

The element e is called the neutral (or identity) element of the group. We can show that in fact it is the neutral element of the group since each group can have only have exactly one neutral element. In the inverse axiom above, we say that there exists y that is the inverse of x. We will show that in fact there is only one inverse corresponding to each element x in the group. The inverse of x is denoted by x^{-1}

Groups are foundational in many different science and math fields such as computer sicnece, mathematics, physics, and chemistry. For cryptography, we use groups to build group based ciphers such as Elliptic Curve cryptography, which is discussed later.

Examples of Groups: $(Z,+,0),(Q,+,0),(R,+,0),(C,+,0) are all groups.$

Group Table using the * operator can be used a look up too. Using group table with two values e, a:

*	e	a
e	e	a
a	a	e

Exercise: Is the following group table correspond to a Abelian Group?

*	e	a
e	e	e
a	a	e

In groups we have the *Uniqueness of a Neutral Element* which says Let $(G,*,e)$ be a group. If $e^{'} \in G$ satisfies the neutral element axiom i.e if for all $x \in G$,

$$e^{'} * x = x = x * e^{'}$$

then

$$e = e^{'}$$

In other words, there exists only one neutral or identity element within a group.

Proof:

Suppose there are two idenity elements $e, e^{'} \in G$

$$e * e^{'} = e^{'} * e = e \text{ Since e is the identity element}$$

Also

$$e^{'} * e = e * e^{'} = e^{'} \text{ Since e' is the identity element}$$

Therefore

$$e = e * e^{'} = e^{'}$$

In groups we also have *Uniqueness of Inverse* whichs says: Let (G, $*$, e) be a group. Let $a \in G$ and $b, b^{'} \in G$ satisfy the inverse axiom for a, i.e.

$$b * a = e = a * b$$

$$b^{'} = a = e = a * b^{'}$$

Then

$$b = b'$$

In other words, there exists only inverse for each element within a group.

Proof:

Let (G, $*$, e) be a group.

Suppose $\exists b, c \in G$ s.t $a * b = e$, $a * c = e$

Then:

b = b * e (since e is the identity element)

= b * (a * c) (since c is an inverse of a)

= (b * a) * c (by associativity axiom of groups)

= e * c (since b is an inverse of a)

= c (since e is the idenity element)

Therefore

$$b = c$$

Exercise: Let (G, $*$, e) be a group. Suppose that G satisfies the folowing: For each x $\in$ G

$$x^2 = e$$

Then (G, $*$, e) is an abelian group.

4.2 Diffie-Hellman Public Exchange

Diffe-Hellman is a form of public key exchange involving groups. By using Alice and Bob, we will show how it works.

Let's say we have a Group, $g \in G$ where g is a huge number.

$g^n =$ e if n = 0, g if n = 1, $g * g^{-1}$ if $n > 1$, $(g^{-1})^{-n}$ if $n < 0$

1. Alice will pick $x_a \in 1, 2, ...G - 1$ Then computes g^{x_a} and sends to Bob.

2. Bob picks $x_b \in 1, 2, ...G - 1$ Then computes g^{x_b} and sends it back to Alice.

3. Alice receives g^{x_b} from Bob and computes $(g^{x_b})^{x_a}$

4. Bob computes $(g^{x_a})^{x_b}$ using what he recieved from Alice.

Alice and Bob agree on a $g^{x_a x_b}$. If it is the key, they must use a diferent type and agree on an encoding method such as $g^{x_a x_b} \to$ int, $\sigma \to$ Z (encoding).

The key is usually used for private or symmetric key encryption communication.

4.3 Elgaml Encryption

Elgaml Encryption is another form of public key exchange using groups. By using Alice and Bob, we will show how it works.

Let's say we have a Group, $g \in G$.

1. Alice picks an x from 1, 2, ... G-1, Alice then computes h = g^x. Alice then publishes h, description of G, G-1, g as the public key, while retaining x as her private key.

2. Bob recieves the information from Alice, he chooses a y from 1, 2, ... G-1 then calculates $f = g^y$. Bob then computes the secret $s = h^y$. Bob then maps his secret message m as an element m' of G. Bob then calculates k = m' $*$ s. Bob sends the ciphertext(f, k) = $(g^y, m' * h^y) = (g^y, m' * (g^x)^y)$ to Alice.

3. Alice then calculates the shared secret by $s = f^x$ and then computes m' = $k * s^{-1}$ which then converst the ciphertext back into the plaintext message m, where s^{-1} is the inverse of s in the gruop G.

The decryption algorithm follows:

$k * s^{-1} = m' * h^y * (g^{xy})^{-1} = m' * g^{xy} * g^{-xy} = m'.$

Chapter 5

Elliptic Curves

5.1 Elliptic Curve Cryptography

Elliptic Curve cryptography is another type of public-key crytography which use the points of an elliptic curve (different than prime numbers seen previously in RSA). It is based upon the unique algebraic structure that elliptic curves have. A difference between Elliptic curve cryptography and RSA is that elliptic curves require much smaller keys, yet are an efficient type of encryption.

Elliptic Curves are used mainly for encryption, digital signatures and even random generators.

Section 5.1 Vocabulary:

Non-Singular Curve is an elliptic curve define by an algebraic expression. If the graph has no cusps or self-intersections, it is said to be non-singular.

Singular Curve is an elliptic curve or point that is not non-singular, if it is, then it is reffered to as singular.

Point at Infinity is the idealized limiting point that is at the 'end' of each line.

Group Properties for Elliptic Curve:

Lets say Group G is for an Elliptic Curve, it then has these laws...

1. The elements of the group are the points of an elliptic curve.

2. The identity element is the point at infinity 0.

3. The inverse of a point p is the symmetric over the x-axis.

4. Addition follows the rules: Given three aligned, non-zero points P, Q, R, their sum is $P + Q + R = 0$. (Note since addition is associative and commutative, it means it's an abelian group).

Example of Addition Law:

Using a group of an elliptic curve which has the form $y^2 = x^3 + ax + b$ has a set of rational points denoted as K, which includes the single point at infinity.

The law follows: Take 2 rational points K, P, Q. We can then 'draw' a straight line through both points and then compute the third point of intersection R (Which is also a rational point). Then we get:

$$P + Q + R = 0$$

Which gives us the point at infinity.

Rules for Adding:

With $P(x_1, y_1), Q(x_2, y_2)$

$x_3 = s_2 - x_1 - x_2 \pmod p and y_3 = s(x_1 - x_3) - y_1 \pmod p$

where

$s = (y_2 - y_1)/x_2 - x_1 \pmod p; if P \neq Q : pointaddition$

$(3x_1^2 + a)/2y_1 \pmod p; if P = Q : pointdoubling$

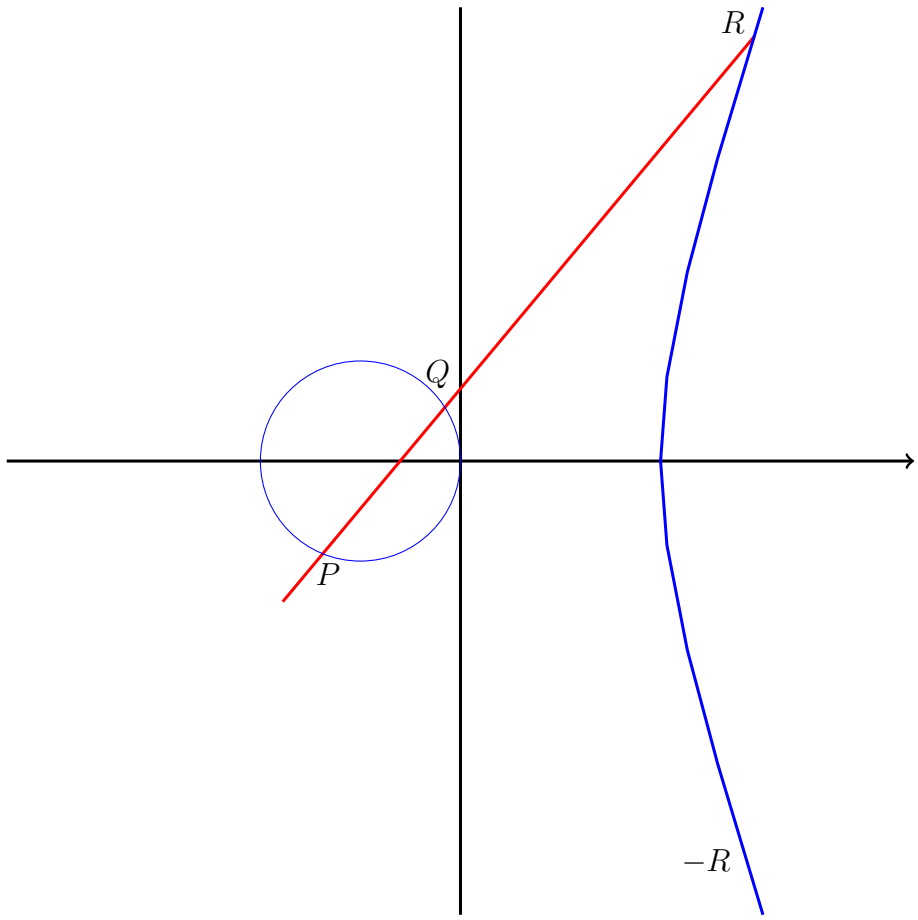
Example:

Given the curve $y^2 = x^3 - 7x$ and points P(-2.35, -1.86) and Q(-0.1, 0.836), $P \neq -Q$ The line will intersect the elliptic curve in exactly one point, -R. By reflecting -R over the x-axis allows us to arrive at point R.

$$P + Q = R$$

$$P\ (-2.35, -1.86) + Q\ (-0.1, 0.836) = -R\ (3.89, 5.62)$$

$$R = (3.89, -5.62)$$


R
Q
P
−R

Index