Crypto Assignment 1

Name: CHE Yulin, Student#: 20292673

Q1 Euclidean Algorithm For greatest common divider(gcd)

```
def gcd_euclidean(lhs, rhs):
    if rhs == 0:
        return lhs
    else:
        return gcd_euclidean(rhs, lhs % rhs)
```

Q1 Extended Euclidean Algorithm

- Implementation:

```
class ExtendedGcdEuclidean:
    def __init__(self, lhs, rhs):
        self.r_list = list([lhs, rhs])
        self.q_list = list([None, None])
        self.x_list = list([1, 0])
        self.y_list = list([0, 1])
        self.iter_list = list([-1, 0])
        self.is_break = False;
        self.compute_final_result()

    def do_one_iteration(self):
        next_tail_index = len(self.iter_list)
        self.iter_list.append(self.iter_list[next_tail_index - 1] + 1)
        self.q_list.append(self.r_list[next_tail_index - 2] /
                           self.r_list[next_tail_index - 1])
        self.r_list.append(self.r_list[next_tail_index - 2] %
                           self.r_list[next_tail_index - 1])
        if self.r_list[next_tail_index] == 0:
            self.is_break = True
            return
        self.x_list.append(
                self.x_list[next_tail_index - 2] -
                self.q_list[next_tail_index] * self.x_list[next_tail_index - 1])
        self.y_list.append(
                self.y_list[next_tail_index - 2] -
                self.q_list[next_tail_index] * self.y_list[next_tail_index - 1])

    def compute_final_result(self):
        while not self.is_break:
            self.do_one_iteration()
```

- Usage:

```
def test_extended_gcd_euclidean(lhs, rhs):
    extend_euclidean_algo = ExtendedGcdEuclidean(lhs, rhs)
    for i in range(0, len(extend_euclidean_algo.iter_list) - 1):
        print 'iter:' + str(extend_euclidean_algo.iter_list[i]) + '\t\tr:' + \
              str(extend_euclidean_algo.r_list[i]) + '\t\tq:' + \
              str(extend_euclidean_algo.q_list[i]) + '\t\tx:' + \
              str(extend_euclidean_algo.x_list[i]) + '\t\ty:' + \
              str(extend_euclidean_algo.y_list[i])

    i = len(extend_euclidean_algo.iter_list) - 1
    print 'iter:' + str(extend_euclidean_algo.iter_list[i]) + '\t\tr:' + \
          str(extend_euclidean_algo.r_list[i]) + '\t\tq:' + \
          str(extend_euclidean_algo.q_list[i])


test_extended_gcd_euclidean(1759, 550)
print '\n'
test_extended_gcd_euclidean(1137, 29)
```

- Result:

i	r	q	x	y
iter:-1	r:1137	q:None	x:1	y:0
iter:0	r:29	q:None	x:0	y:1
iter:1	r:6	q:39	x:1	y:-39
iter:2	r:5	q:4	x:-4	y:157
iter:3	r:1	q:1	x:5	y:-196
iter:4	r:0	q:5	None	None

- Answer of Q1:
  the multiplicative inverse of (29 modulo 1137) is -196, since (29 * (-196)) % 1137 = 1

Q2 Transposition Cipher & Substitution Cipher

Substitution Cipher, includes a one-to-one mapping-function from the english letters domain to english letters domain. So it is possible to do some statistical analysis, based on the common frequency distribution of english letters.
And in Q2, we know that 20000 letters are quite enough to conduct such analysis. If we apply this frequency pattern analysis on the cipher-text,
i.e, find a one-to-one mapping-function according to the frequency statistical data, the text which is encrypted through subsitution cipher, should be readable after being decrypted by the mapping function.

Whereas the above frequency analysis methodology does not fit for the crack of transposition cipher, since according to the definition of transposition cipher, it just simply transpose the whole plaintext without any substitution.

So, in conclusion, if after we apply the above methodology on the 20000-length text, the text becomes much more readable, then it is substitution cipher, else it is transposition cipher.

Q3 Answer

- method
  - model to measure the messy criteria, link: http://quipqiup.com/index.php
  - the frequency analysis tool, link: http://cryptoclub.org/tools/crack_substitutioncipher.php
- one-to-one mapping function, i.e, symmetric key:

a	b	c	d	e	f	g	h	i	j	k	l	m
Y	L	X	U	W	Z	C	A	D	E	F	B	G

n	o	p	q	r	s	t	u	v	w	x	y	z
I	V	H	O	K	R	Q	P	T	N	J	M	S

- original text:

> a legacy of atmospheric atomic bomb testing is present in an unlikely place: people's teeth. according to a report published today in the journal nature, bomb-generated carbon isotopes trapped in tooth enamel may provide a more precise method for determining a deceased individual's age than other forensic method scan.
>
> the above ground nuclear tests that occurred between 1955 and 1963 dramatically increased the amount of the isotope carbon 14 in the atmosphere. the levels rapidly equalized around the globe, even though the explosions occurred at only a few locations, and entered plants in the food chain through photosynthesis. by eatingp lants, and animals that feed on plants, humans absorb carbon 14 and exhibit levelsof the benign, traceable isotope that aresimilar toatmospheric concentrations. whatismore, carbon 14 decays witha half-life of 5,730 years, a phenomenon that scientists can exploit as away to determine the ages of objects that contain the isotope. for the new study, jonasfrisen of the medical nobel institute in stockholm, sweden, and his colleagues analyzed the carbon content of toothe namel. because teeth do not exhibit any turn over during a person's life, the scientists can determine when a tooth formed by comparing its carbon 14 content to past atmospheric levels. in addition, adult teeth form during a distinct period of childhood development around age 12, so this information can be translated into the age of an individual.

Q4 Substitution Cipher

- Implementation

```
class EngIndexBiMap:
    def __init__(self):
        self.char2index_dict = dict()
        self.index2char_dict = dict()
        for i in range(0, 25):
            self.char2index_dict[chr(ord('a') + i)] = i
            self.index2char_dict[i] = chr(ord('a') + i)

    def char2index(self, ch):
        return self.char2index_dict[ch]

    def index2char(self, index):
        return self.index2char_dict[index]


# val from 0 to 25
def encrypt(letter_index, k0, k1):
    return (letter_index * k0 + k1) % 26


def cipher(msg, k0, k1):
    my_bimap = EngIndexBiMap()
    cipher_text = list()
    for ele in msg:
        cipher_text.append(my_bimap.index2char(
            encrypt(my_bimap.char2index(ele), k0, k1)))
    return cipher_text
```

- Usage

```
print cipher('lecture', 3, 1)
```

- Result

```
['i', 'n', 'h', 'g', 'j', 'a', 'n']
```

- Answer of Q2: `inhgjan`

Q5 Answer

- Implementation of Finite Field

```python
# Finite Field 2^8 where p(x) = x^8 + x^4 + x^3 + x + 1
class FiniteFieldNumber:
    # p(x) = x^8 + x^4 + x^3 + x + 1
    magical_number = int('100011011', 2)

    def __init__(self, value, is_bin=True):
        if is_bin:
            self.integer_32bits = int(value, 2)
        else:
            self.integer_32bits = value

    def __div__(self, other):
        ret_integer_num = int(0)
        tmp_finite_field_num = FiniteFieldNumber(self.integer_32bits, False)
        while tmp_finite_field_num > other:
            tmp_whole_len = len(bin(tmp_finite_field_num.integer_32bits))
            other_whole_len = len(bin(other.integer_32bits))
            tmp_finite_field_num = tmp_finite_field_num - FiniteFieldNumber(
                other.integer_32bits << (tmp_whole_len -
                                         other_whole_len), False)
            ret_integer_num ^= 1 << (tmp_whole_len - other_whole_len)
        return FiniteFieldNumber(ret_integer_num, False)

    def __lt__(self, other):
        return self.integer_32bits < other.integer_32bits

    def __mod__(self, other):
        tmp_finite_field_num = FiniteFieldNumber(self.integer_32bits, False)
        while tmp_finite_field_num > other:
            tmp_whole_len = len(bin(tmp_finite_field_num.integer_32bits))
            other_whole_len = len(bin(other.integer_32bits))
            tmp_finite_field_num = tmp_finite_field_num - FiniteFieldNumber(
                other.integer_32bits << (tmp_whole_len -
                                         other_whole_len), False)
        return tmp_finite_field_num

    # multiplication on GF(2^8)
    def __mul__(self, other):
        bin_str = bin(self.integer_32bits)
        whole_len = len(bin_str)
        new_int32 = int(0)
        for index in range(0, whole_len):
            order_num = whole_len - 1 - index
            if (bin_str[index]) == '1':
                new_int32 ^= other.integer_32bits << order_num
        return FiniteFieldNumber(new_int32, False) % \
               FiniteFieldNumber(FiniteFieldNumber.magical_number, False)

    def __add__(self, other):
        return FiniteFieldNumber(self.integer_32bits ^
                                 other.integer_32bits, False)

    def __sub__(self, other):
```

```
        return FiniteFieldNumber(self.integer_32bits ^
                                 other.integer_32bits, False)

    def __str__(self):
        bin_str = bin(self.integer_32bits)
        whole_len = len(bin_str)
        ret_str = str()
        start_flag = False
        for index in range(0, whole_len):
            order_num = whole_len - 1 - index
            if (bin_str[index]) == '1':
                if start_flag:
                    ret_str += ' + '
                else:
                    start_flag = True
                ret_str += 'x^' + str(order_num)
        return ret_str


if __name__ == '__main__':
    magical_number = FiniteFieldNumber(FiniteFieldNumber.magical_number, False)
    print 'p(x): ' + str(magical_number)

    number2 = FiniteFieldNumber('0')
    number3 = FiniteFieldNumber('1000110')
    print 'Q5-(1):' + str(number2 - number3)

    number0 = FiniteFieldNumber('1000110')
    number1 = FiniteFieldNumber('10001011')
    print 'Q5-(2):' + str(number0 + number1)

    print 'Q5-(3):' + str(number0 * number1)

    number4 = FiniteFieldNumber('100001111111010')
    print number4 / magical_number
```

- Result for first three questions

```
Q5-(1):x^6 + x^2 + x^1
Q5-(2):x^7 + x^6 + x^3 + x^2 + x^0
Q5-(3):x^7 + x^5 + x^3 + x^2
```

- Answer for first three questions of Q5

```
Q5-(1):x^6 + x^2 + x^1
Q5-(2):x^7 + x^6 + x^3 + x^2 + x^0
Q5-(3):x^7 + x^5 + x^3 + x^2
```

- Implementation of Q5(4)

```
class ExtendedGcdEuclidean:
    def __init__(self, lhs, rhs):
        self.r_list = list([lhs, rhs])
        self.q_list = list([None, None])
        self.x_list = list([FiniteFieldNumber(1, False),
                            FiniteFieldNumber(0, False)])
        self.y_list = list([FiniteFieldNumber(0, False),
                            FiniteFieldNumber(1, False)])
        self.iter_list = list([-1, 0])
        self.is_break = False;
        self.compute_final_result()

    def do_one_iteration(self):
        next_tail_index = len(self.iter_list)
        self.iter_list.append(self.iter_list[next_tail_index - 1] + 1)
        self.q_list.append(self.r_list[next_tail_index - 2] /
                           self.r_list[next_tail_index - 1])
        self.r_list.append(self.r_list[next_tail_index - 2] %
                           self.r_list[next_tail_index - 1])
        if self.r_list[next_tail_index].integer_32bits == 0:
            self.is_break = True
            return
        self.x_list.append(self.x_list[next_tail_index - 2] -
                           self.q_list[next_tail_index] *
                           self.x_list[next_tail_index - 1])
        self.y_list.append(self.y_list[next_tail_index - 2] -
                           self.q_list[next_tail_index] *
                           self.y_list[next_tail_index - 1])

    def compute_final_result(self):
        while not self.is_break:
            self.do_one_iteration()


def test_extended_gcd_eculidean(lhs, rhs):
    extend_euclidean_algo = ExtendedGcdEuclidean(lhs, rhs)
    for i in range(0, len(extend_euclidean_algo.iter_list) - 1):
        print 'iter:' + str(extend_euclidean_algo.iter_list[i]) + '\t\tr:' + \
              str(extend_euclidean_algo.r_list[i]) + '\t\tq:' + \
              str(extend_euclidean_algo.q_list[i]) + '\t\tx:' + \
              str(extend_euclidean_algo.x_list[i]) + \
              '\t\ty:' + str(extend_euclidean_algo.y_list[i])

    i = len(extend_euclidean_algo.iter_list) - 1
    print 'iter:' + str(extend_euclidean_algo.iter_list[i]) + '\t\tr:' \
          + str(extend_euclidean_algo.r_list[i]) + '\t\tq:' \
          + str(extend_euclidean_algo.q_list[i])


if __name__ == '__main__':
    px_number = FiniteFieldNumber(FiniteFieldNumber.magical_number, False)

    ax_number0 = FiniteFieldNumber('10000011')
```

```
test_extended_gcd_eculidean(px_number, ax_number0)

print '\n'

ax_number1 = FiniteFieldNumber('1000110')
test_extended_gcd_eculidean(px_number, ax_number1)
```

- Result of Q5(4)

i	r	q	x	y
iter:-1	r:x^8 + x^4 + x^3 + x^1 + x^0	q:None	x:x^0	y:0
iter:0	r:x^6 + x^2 + x^1	q:None	x:0	y:x^0
iter:1	r:x^1 + x^0	q:x^2	x:x^0	y:x^2
iter:2	r:x^1	q:x^5 + x^4 + x^3 + x^2	x:x^5 + x^4 + x^3 + x^2	y:x^7 + x^6 + x^5 + x^4 + x^0
iter:3	r:x^0	q:x^0	x:x^5 + x^4 + x^3 + x^2 + x^0	y:x^7 + x^6 + x^5 + x^4 + x^2 + x^0
iter:4	r:0	q:x^1	None	None

- Answer of Q5(4)
  So, the multiplicative inverse c(x) of a(x) is `x^7 + x^6 + x^5 + x^4 + x^2 + x^0`