Reid Nagurka

HW 4

Task 1

1) #below is one_time_pad.py

```
#!/usr/bin/python2.7
import sys

def oneTimePad(otp, file):
    #read contents from file
    f = open(file, "r")
    contents = f.read()
    print(contents)

    #if the key is shorter than the filename, repeat the key until it is the same size
    while(len(otp) < len(contents)):
        otp += otp
    #call the encryp function, it returns the encrypted text
    ciphertext = encrypt(contents, otp)
    print(ciphertext)
     #to test, xor the key with the ciphertext, this should result in the original plaintext content
    plaintext = encrypt(ciphertext, otp)
    print(plaintext)

#key has already been mutated to have the size necessary to compute
def encrypt(text, key):
    #get the integer ASCII value for each character in the file and for each character in the key
    text_array = []
    key_array = []
    for each in text:
```

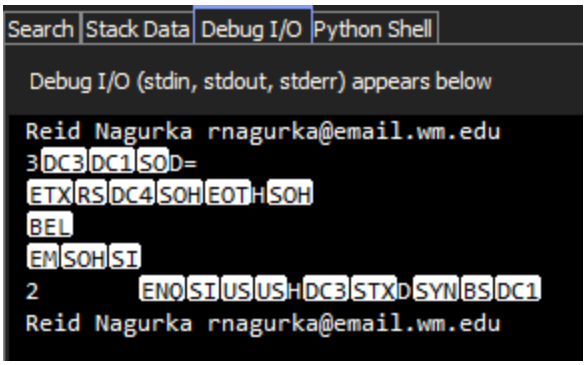
```
        text_array.append(ord(each))


    for each in key:
        key_array.append(ord(each))


    #for the length of the file encrypting, xor the characters at the same index of the file content and the key
    ciphertextarray = []
    idx = 0
    while(idx < len(text_array)):
        ciphertextarray.append(text_array[idx] ^ key_array[idx])
        idx += 1
    #ciphertextarray now stores the xored int values


    #convert each element in ciphertextarray to a character, then concatenate into string
    ciphertext = ""
    for each in ciphertextarray:
        ciphertext += str(unichr(each))
    return ciphertext


#key = "avxjdsldkfjehsdfjlsdkrlhnvsfdojsldfsifysdfjosduhfpofjpwehjbksdjfbksf"
#filename = "one_time_pad_file_to_encrypt.txt"
if len(sys.argv) != 3:
            exit('usage ./rot.py k f_name')
key = sys.argv[1]
#print(key)
filename = sys.argv[2]
oneTimePad(key, filename)
```



Picture to the left shows the input file first, then the encrypted text, then the restored text (reversing the encryption process using same key).

2) It is not secure because the key is used multiple times because each student uses the same key. This violates a property of the one-time-pad key. If each student was given a unique, randomly generated key, it would be completely secure to post over Piazza. (double check / elaborate why the property is broken and what that means)

Task 2

1) Method 1: Brute force attack. We know the possible rotation key is a value between 0 and 255, so we can brute force try every value until the output is logical.
   Method 2: Frequency analysis. Certain characters are more common than others (a, e), only certain letters appear frequently next to each other (ee, ll), and certain small words are common (we, she, he). Using this information, we can determine the most common characters in the ciphertext and map them to the most common letters in English and analyze the result.
2) #below is rot.py. NOTE: the brute force method is commented out here for simpler testing

```
#!/usr/bin/python2.7

import sys, base64, re
input_base64 = False

def decrypt(key):
 c = ''
 for i in s:
  c += chr(ord(i) ^ key % 256)
 #print "key: ", key

 if input_base64: # print plaintext
  print c

 else:
  c_64 = base64.b64encode(c)
  print c_64
```

```
def frequencyAnalysisDecrypt():
  #basic idea: look for common characters in ciphertext to map to common characters in plaintext
  #we know 'e' is most common English character, so try to match that with a ciphertext char
  #we also know 'ee' frequently appears in English. Update: need to discount the most frequent character it maps to' ', not an English letter. I discount this b/c not everyone uses '  ' for after periods and to cover a text that has no spaces in English

  #NOTE: Checking for repeated chars is not necessary with this text, but just makes the algorithm stronger.

  #create dictionary of all characters in ciphertext with count of frequency
  cipherdic = {}
  #repeatdic will hold all two character repetitions and their frequency
  repeatdic = {}
  idx = 0
  #ch will hold each two repeated chars next to each other
  ch = ""

  for i in s:
    #i will be key of dictionary
    #if not in dictionary, create entry and set count to 1
    if i not in cipherdic:
      cipherdic[i] = 1
    #else, update increment count
    else:
      cipherdic[i] += 1
    #bounds checking
    if(idx < len(s) - 1):
      #if two adjacent chars are next to each other, add to repeatdic (same way as in cipherdic)
      #use ch as dictionary key for repeatdic
      if (s[idx] == s[idx + 1]):
        ch = s[idx] + s[idx + 1]
        if ch not in repeatdic:
          repeatdic[ch] = 1
        else:
          repeatdic[ch] += 1
    idx += 1
  #reverse order sort cipherdic and repeatdic to get a list of most frequent chars (most frequent -> least frequent order)
  sorted_dic = sorted(cipherdic.items(), key=lambda x: x[1], reverse=True)
  #remove most frequent character in sorted_dic b/c it will map to ' ' in plaintext.
  sorted_dic.pop(0)
  sorted_repeat_dic = sorted(repeatdic.items(), key=lambda x: x[1], reverse=True)
```

```
  #both will be a list of all characters that are in the top five frequency that appear in both sorted_dic and sorted_repeat_dic. Just top 5 is used for simplicity (could include more but the letter that maps to 'e' should be among the top 5 most common characters)

  both = []
  #loop counters
  x = 0
  while(x < 6):
    y = 0
    while(y < 6):
      #if there is a single frequency match and a double frequency match, add to list
      if sorted_dic[x][0] == sorted_repeat_dic[y][0][:1]:
        both.append(sorted_dic[x][0])
      y += 1
    x += 1
  #both is naturally sorted using the looping above. This means that it is sorted using precedence of appearing more frequent by itself is more important than frequency of appearing repeated.
  #remove the first element b/c it will map to ' '


  #first element should be mapped to e
  key = ord(both[0]) ^ ord('e') % 256
  decrypt(key)

  return


#for rot we don't know key, so just want the filename
if len(sys.argv) != 2:
        exit('usage ./rot.py k f_name')
#key = int(sys.argv[1])
f_name = sys.argv[1]

#f_name = "ciphertext_rot.txt"
s = open(f_name).read()
#detect base 64
if re.match('^([A-Za-z0-9+/]{4})*([A-Za-z0-9+/]{3}=|[A-Za-z0-9+/]{2}==)?$', s):
  input_base64 = True
  s = base64.b64decode(s)

#brute force:
#for key in range(0, 256):
#  decrypt(key)
```

#frequency analysis
frequencyAnalysisDecrypt()

```
key:  22

Call me Ishmael.  Some years ago  never mind how long
precisely  having little or no money in my purse, and nothing
particular to interest me on shore, I thought I would sail about a
little and see the watery part of the world.  It is a way I have of
driving off the spleen and regulating the circulation.  Whenever I
find myself growing grim about the mouth; whenever it is a damp,
drizzly November in my soul; whenever I find myself involuntarily
pausing before coffin warehouses, and bringing up the rear of every
funeral I meet; and especially whenever my hypos get such an upper
hand of me, that it requires a strong moral principle to prevent me
from deliberately stepping into the street, and methodically knocking
people's hats off  then, I account it high time to get to sea as soon
as I can.  This is my substitute for pistol and ball.  With a
philosophical flourish Cato throws himself upon his sword; I quietly
take to the ship.  There is nothing surprising in this.  If they but
knew it, almost all men in their degree, some time or other, cherish
very nearly the same feelings towards the ocean with me.
```

Task 3

1)
   a. In order to find the key length, look for repeated patterns in ciphertext. The repeated patterns should be a fixed distance apart. That distance means the key is likely a factor of that distance.
   b. For example, if the distance between a repeated string of characters is 15, then the key is likely 15, 5 or 3 characters long. The likelihood increases the more frequent the pattern reoccurs.
   c. Record all distances between all repeated trigrams with their corresponding frequency. That list of distances with frequencies will show likely candidates for the key length. If the top few results have similar frequencies and all of them are factors of one of the other (for example, if the frequent distances are 2, 4, and 8), then the key length is most likely the largest factor (in this case, 8).
   d. Once the length of the key is known (n), separate the ciphertext into different strings with characters spaced n apart in the original ciphertext. In other words, take every n characters and add it to a string. Repeat n total times (so there are n strings).
   e. Run frequency analysis on each string generated from part d. Each analysis will give one character of the key (the first string will give the first character of the key).
2) #below is vcrypt.py

#!/usr/bin/python2.7

```
import sys, base64, re
input_base64 = False

def encr(text, key):
  key_len=len(key)
  l = [text[i:i+key_len] for i in xrange(0,len(text), key_len)]
  r = ''
  for i in l:
    if len(i) < key_len: #padding needed
       i = i + ' '  * (key_len - len(i))
    t=''
    for n, j in enumerate(i):
      t+=chr(ord(j) ^ ord(key[n]))
    assert(len(t) == key_len)
    r+=t
  return r

#gets a list of factors of a number
def factors(n):
  l = []
  #we don't care about 1, so start at 2
  for i in range(2,n):
    if n % i == 0:
      l.append(i)
  return l

def getKeyLength(s):
  #dic will have: (ch as key, then frequency of appearence in text
```

```
 dic = {}
 #factorList will have a list of all factors of the difference in distances between two matching three character strings
 factorList = []

 for i in range (0, len(s) - 3):
  #get the first three characters
  ch1 = s[i:i+3]
  #compare those first three to every other set of three
  for j in range(i+1, len(s) - 3):
   ch2 = s[j:j+3]
   if(ch1 == ch2):
    #add the factors of the difference in distance between the character occurences
    factorList.extend(factors(j-i))

 #populate dic with the frequency of occurence of each factor calculated above
 for x in factorList:
  if x not in dic:
   dic[x] = 1
  else:
   dic[x] += 1

 #sort the dictionary in reverse order so factor, frequency is in descending order
 newList = sorted(dic.items(), key=lambda x: x[1], reverse=True)

 #now have a list of: (factors, frequency)
 '''
```

find where the value is no longer a multiple of the previous ones. For example, in this case newList goes: 2, 4, 8, 16, 32, 3, 6, etc. This means we want the index of 16 b/c it is a multiple of the previous elements and within 2-20 key length restriction. We could look at the frequency too to be sure.

```
  '''
  #default key length to the greatest frequency.
  keySize = newList[0][0]
  i = 0
  #while the next value is divisible by the previous
  while (newList[i+1][0] % newList[i][0] == 0):
    #make sure between 2 and 20 characters long
    if (newList[i+1][0] >= 2 and newList[i+1][0] <= 20):
      keySize = newList[i+1][0]
    i += 1

  return keySize



if len(sys.argv) != 2:
  exit('Usage: ./crypt.py "key" input_file_name')
#key = sys.argv[1]
fname = sys.argv[1]



#fname = "ciphertext_vig.txt"
s = open(fname).read()



#detect base 64
if re.match('^([A-Za-z0-9+/]{4})*([A-Za-z0-9+/]{3}=|[A-Za-z0-9+/]{2}==)?$', s):
  input_base64 = True
  s = base64.b64decode(s)
keyLength = getKeyLength(s)
```

```
print "key length is ", keyLength
'''
c = encr(s, key)

if input_base64: # print plaintext
  print c
else:
  c_64 = base64.b64encode(c)
  print c_64
'''
```

```
[03/31/19]seed@VM:~/Documents$ python vcrypt.py ciphertext_vig.txt
key length is  16
```

NOTE: Appendix at end of the pdf has screenshots of all the scripts to make them easier to read. The text is included above so that it can be copied and pasted if needed.

Appendix

one_time_pad.py

```python
#!/usr/bin/python2.7
import sys

def oneTimePad(otp, file):
    #read contents from file
    f = open(file, "r")
    contents = f.read()
    print(contents)

    #if the key is shorter than the filename, repeat the key until it is the same size
    while(len(otp) < len(contents)):
        otp += otp
    #call the encryp function, it returns the encrypted text
    ciphertext = encrypt(contents, otp)
    print(ciphertext)
     #to test, xor the key with the ciphertext, this should result in the original plaintext content
    plaintext = encrypt(ciphertext, otp)
    print(plaintext)


#key has already been mutated to have the size necessary to compute
def encrypt(text, key):

    #get the integer ASCII value for each character in the file and for each character in the key
    text_array = []
    key_array = []
    for each in text:
        text_array.append(ord(each))

    for each in key:
        key_array.append(ord(each))

    #for the length of the file encrypting, xor the characters at the same index of the file content and the key
    ciphertextarray = []
    idx = 0
    while(idx < len(text_array)):
        ciphertextarray.append(text_array[idx] ^ key_array[idx])
        idx += 1
    #ciphertextarray now stores the xored int values

    #convert each element in ciphertextarray to a character, then concatenate into string
    ciphertext = ""
    for each in ciphertextarray:
        ciphertext += str(unichr(each))
    return ciphertext

#key = "avxjdsldkfjehsdfjlsdkrlhnvsfdojsldfsifysdfjosduhfpofjpwehjbksdjfbksf"
#filename = "one_time_pad_file_to_encrypt.txt"
if len(sys.argv) != 3:
        exit('usage ./rot.py k f_name')
key = sys.argv[1]
print(key)
filename = sys.argv[2]
oneTimePad(key, filename)
```

rot.py

```
#!/usr/bin/python2.7

import sys, base64, re
input_base64 = False

def decrypt(key):
  c = ''
  for i in s:
    c += chr(ord(i) ^ key % 256)
  #print "key: ", key

  if input_base64: # print plaintext
    print c

  else:
    c_64 = base64.b64encode(c)
    print c_64

def frequencyAnalysisDecrypt():
  #basic idea: look for common characters in ciphertext to map to common characters in plaintext
  #we know 'e' is most common English character, so try to match that with a ciphertext char
  #we also know 'ee' frequently appears in English. Update: need to discount the most frequent character it maps to' ', ↵
↳ not an English letter. I discount this b/c not everyone uses '  ' for after periods and to cover a text that has no ↵
↳ spaces in English

  #NOTE: Checking for repeated chars is not necessary with this text, but just makes the algorithm stronger.

  #create dictionary of all characters in ciphertext with count of frequency
  cipherdic = {}
  #repeatdic will hold all two character repetitions and their frequency
  repeatdic = {}
  idx = 0
  #ch will hold each two repeated chars next to each other
  ch = ""

  for i in s:
    #i will be key of dictionary
    #if not in dictionary, create entry and set count to 1
    if i not in cipherdic:
      cipherdic[i] = 1
    #else, update increment count
    else:
      cipherdic[i] += 1
    #bounds checking
    if(idx < len(s) - 1):
      #if two adjacent chars are next to each other, add to repeatdic (same way as in cipherdic)
      #use ch as dictionary key for repeatdic
      if (s[idx] == s[idx + 1]):
        ch = s[idx] + s[idx + 1]
        if ch not in repeatdic:
          repeatdic[ch] = 1
        else:
          repeatdic[ch] += 1
    idx += 1
  #reverse order sort cipherdic and repeatdic to get a list of most frequent chars (most frequent -> least frequent order)
  sorted_dic = sorted(cipherdic.items(), key=lambda x: x[1], reverse=True)
  #remove most frequent character in sorted_dic b/c it will map to ' ' in plaintext.
  sorted_dic.pop(0)
  sorted_repeat_dic = sorted(repeatdic.items(), key=lambda x: x[1], reverse=True)

  #both will be a list of all characters that are in the top five frequency that appear in both sorted_dic and ↵
↳ sorted_repeat_dic. Just top 5 is used for simplicity (could include more but the letter that maps to 'e' should be ↵
↳ among the top 5 most common characters)

  both = []
  #loop counters
  x = 0
  while(x < 6):
    y = 0
```

```
65      y = 0
66      while(y < 6):
67        #if there is a single frequency match and a double frequency match, add to list
68        if sorted_dic[x][0] == sorted_repeat_dic[y][0][:1]:
69          both.append(sorted_dic[x][0])
70        y += 1
71      x += 1
72    #both is naturally sorted using the looping above. This means that it is sorted using precedence of appearing more
      frequent by itself is more important than frequency of appearing repeated.
73    #remove the first element b/c it will map to ' '
74
75
76    #first element should be mapped to e
77    key = ord(both[0]) ^ ord('e') % 256
78    decrypt(key)
79
80    return
81
82
83  #for rot we don't know key, so just want the filename
84  if len(sys.argv) != 2:
85          exit('usage ./rot.py k f_name')
86  #key = int(sys.argv[1])
87  f_name = sys.argv[1]
88
89  #f_name = "ciphertext_rot.txt"
90  s = open(f_name).read()
91  #detect base 64
92  if re.match('^([A-Za-z0-9+/]{4})*([A-Za-z0-9+/]{3}=|[A-Za-z0-9+/]{2}==)?$', s):
93    input_base64 = True
94    s = base64.b64decode(s)
95
96  #brute force:
97  #for key in range(0, 256):
98  #  decrypt(key)
```

```
96  #brute force:
97  #for key in range(0, 256):
98  #  decrypt(key)
99
100 #frequency analysis
101 frequencyAnalysisDecrypt()
```

vcrypt.py

```python
#!/usr/bin/python2.7

import sys, base64, re
input_base64 = False

def encr(text, key):
  key_len=len(key)
  l = [text[i:i+key_len] for i in xrange(0,len(text), key_len)]
  r = ''
  for i in l:
    if len(i) < key_len: #padding needed
      i = i + ' '  * (key_len - len(i))
    t=''
    for n, j in enumerate(i):
      t+=chr(ord(j) ^ ord(key[n]))
    assert(len(t) == key_len)
    r+=t
  return r

#gets a list of factors of a number
def factors(n):
  l = []
  #we don't care about 1, so start at 2
  for i in range(2,n):
    if n % i == 0:
      l.append(i)
  return l

def getKeyLength(s):
  #dic will have: (ch as key, then frequency of appearence in text
  dic = {}
  #factorList will have a list of all factors of the difference in distances between two matching three character strings
  factorList = []

  for i in range (0, len(s) - 3):
    #get the first three characters
    ch1 = s[i:i+3]
    #compare those first three to every other set of three
    for j in range(i+1, len(s) - 3):
      ch2 = s[j:j+3]
      if(ch1 == ch2):
        #add the factors of the difference in distance between the character occurences
        factorList.extend(factors(j-i))

  #populate dic with the frequency of occurence of each factor calculated above
  for x in factorList:
    if x not in dic:
      dic[x] = 1
    else:
      dic[x] += 1

  #sort the dictionary in reverse order so factor, frequency is in descending order
  newList = sorted(dic.items(), key=lambda x: x[1], reverse=True)

  #now have a list of: (factors, frequency)
  '''
  use idx to find the index of the list where the value (first element in each i)is no longer a multiple of the previous ones. For example, in this case it goes: 2, 4, 8, 16, 32, 3, 6, etc. This means we want the index of 16 b/c it is a multiple of the previous elements and within 2-20 key length restriction. We could look at the frequency too to be sure.
  '''
  #default key length to the greatest frequency.
  keySize = newList[0][0]
  i = 0
  #while the next value is divisible by the previous
  while (newList[i+1][0] % newList[i][0] == 0):
    #make sure between 2 and 20 characters long
    if (newList[i+1][0] >= 2 and newList[i+1][0] <= 20):
      keySize = newList[i+1][0]
```

```
  while (newList[i+1][0] % newList[i][0] == 0):
    #make sure between 2 and 20 characters long
    if (newList[i+1][0] >= 2 and newList[i+1][0] <= 20):
      keySize = newList[i+1][0]
    i += 1

  return keySize


if len(sys.argv) != 2:
  exit('Usage: ./crypt.py "key" input_file_name')
#key = sys.argv[1]
fname = sys.argv[1]


#fname = "ciphertext_vig.txt"
s = open(fname).read()


#detect base 64
if re.match('^([A-Za-z0-9+/]{4})*([A-Za-z0-9+/]{3}=|[A-Za-z0-9+/]{2}==)?$', s):
  input_base64 = True
  s = base64.b64decode(s)
keyLength = getKeyLength(s)
print "key length is ", keyLength
'''
c = encr(s, key)

if input_base64: # print plaintext
  print c
else:
  c_64 = base64.b64encode(c)
  print c_64
'''
```