William Hupp
CIS5930

Crypto I Write-Up

Entropy I:

20 points

Problem:

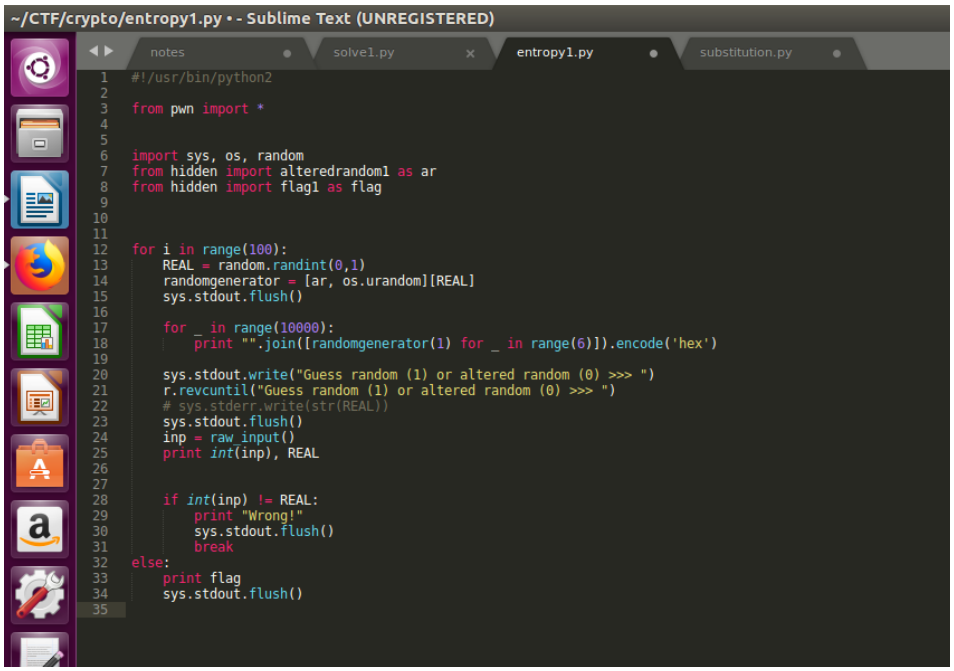
Devise a function to measure entropy in a bit string. To do this, think of the different desirable traits for a random string (about the same number of 1's and 0's, no long runs of 1's and 0's, unpredictability) and create some mechanism to score strings. It should operate on arbitrary-length bitstrings. Describe your thought process and code in your writeup. Feel free to discuss which aspects of entropy your function measures in the class chat room at n0l3ptr.slack.com in the #ctfclass chat

Note: Entropy(0b11111111) should equal Entropy(0b00000000), and both should be less than actual random bitstrings you generate.

Use your entropy function to automatically decide if a set of strings is from a random number generator or has been tampered with in any way
nc crypto.n0l3ptr.com 8001

Solution:

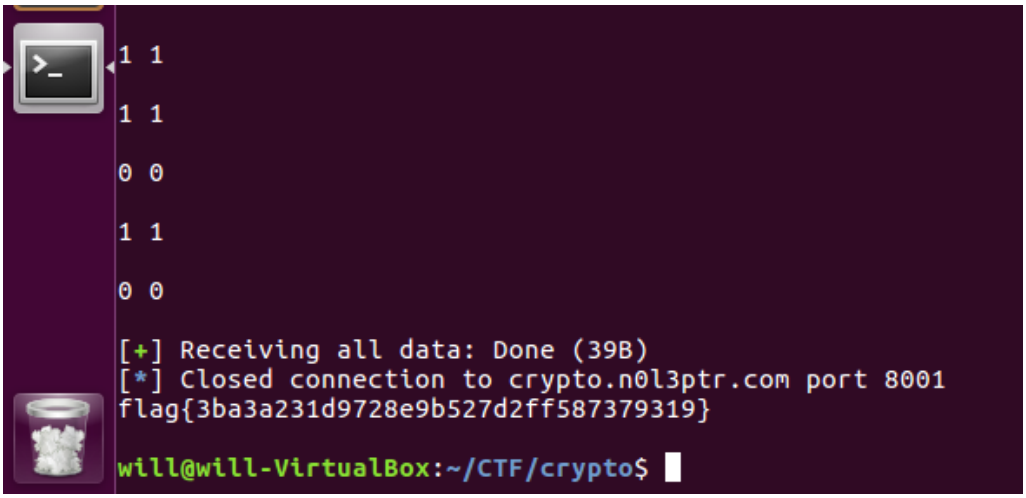
I started by analyzing the provided source code ‘entropy1.py’.



Here we see there is a loop that runs 100 times. It generates 10,000 random numbers between 0 and 1 and encodes them into 6 bytes of hex. It prints the hex to standard output and prompts the user to guess to see if a number is random or if it has been altered by entering 1 and 0 respectively. We must correctly determine if the number is random or altered 100 times for the flag to print. We know this because the flag is printed after the for loop runs 100 times without exiting. Incorrect answers for determining randomness lead the program to exit. This was all confirmed through running the program with 'nc crypto.n0l3ptr.com 8001'. To get the correct randomness of the randomly generated number, we must calculate the entropy. I solved this by writing a Python script which is attached as 'solve1.py'.

The first two lines import the math and pwn Python libraries. The third line connects to the server where the source code is stored. I then create a while loop that runs 100 times. The program on the server displays lines of 6 byte hex and then prompts to guess the entropy with 'Guess random (1) or altered random (0) >>> '. So, I receive all the bytes up until 'Guess'. I then eliminate 'Guess' and '\n' and spaces from the string received, so I have a long string of hexadecimal. I then read until where I am prompted to input 1 or 0 with pwntools (so I can send input to the server). I convert the hex string to actual hex, then binary, and then I convert the binary into a binary string so I can iterate over it to count the number of zeroes and ones. This count of ones vs zeroes will be used to determine entropy. I start the count for 1 at -1 because there is a 'b' in my string of hex that doesn't get translated (indicated by 0b(000000)). My logic counts that 'b' as a 1. I then loop through the input and count the occurrences of 0's and 1's and add to their respective counters. I proceed to convert the binary counters to floats so that I am not performing integer math and get a decimal division result. As per Mitch's suggestion in class, if the percentage of 1's to 0's is greater than or equal to .99, then the integer is random. Otherwise it's not. I tried all sorts of values, including count1/count0 > 1 but that did not work. I also originally tried a substring method for determining entropy but that did not work. I send the appropriate

results to the server (0 or 1) based off the calculation determined by comparing if the percentage of 1's to 0's is greater than or equal to .99 I receive input from the server and loop again in the the while loop. At the end of the while loop, I print all output from the server (the flag).



flag{3ba3a231d9728e9b527d2ff587379319}

Entropy 2:

20 points

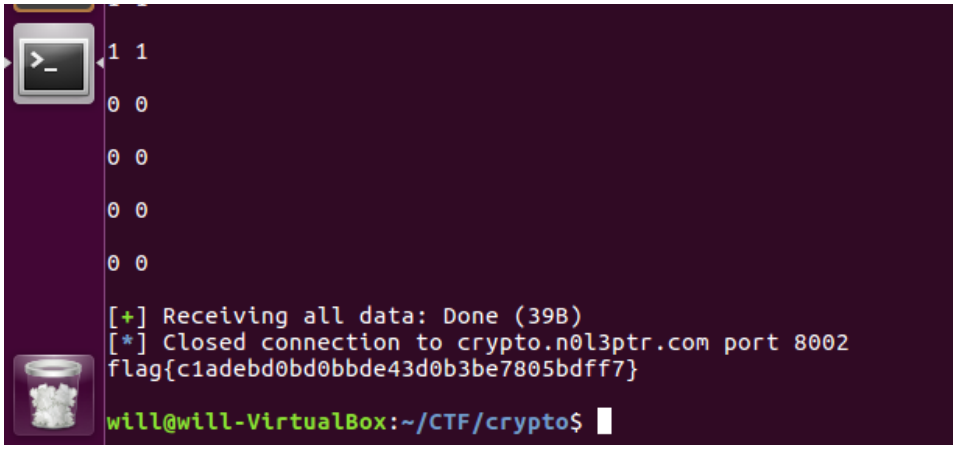
Problem:

Use your entropy function from the previous question to solve a similar question.

nc crypto.n0l3ptr.com 8002

Solution:

I used the same script as for Entropy 1. All I did was change the port when connecting to the server from 8001 to 8002. The code is attached as 'script2.py'



flag{c1adebd0bd0bbde43d0b3be7805bdff7}

Entropy 3:

10 points

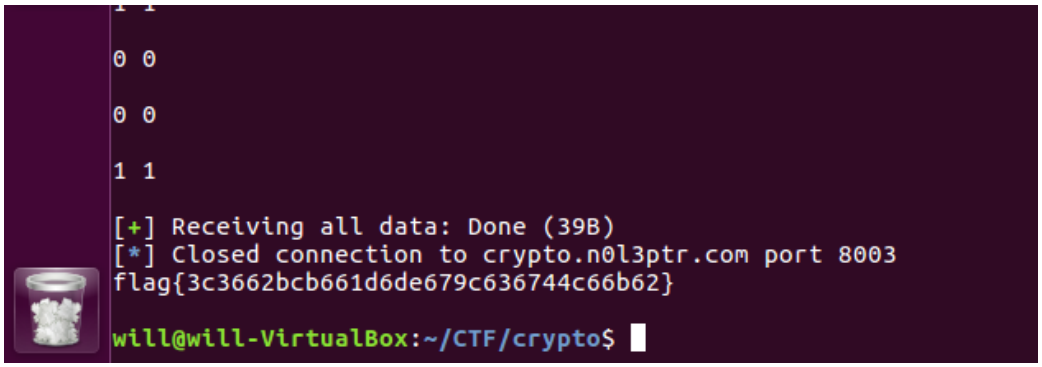
Problem:

Use your entropy function from the previous question to solve a similar question.

nc crypto.n0l3ptr.com 8003

Solution:

I used the same script as for Entropy 1. All I did was change the port when connecting to the server from 8001 to 8003. The code is attached as ‘script3.py’



flag{3c3662bcb661d6de679c636744c66b62}

Entropy Extra Credit:

In entropy 1 and 2 the user is asked to determine if a number is random. This is solved through comparing the number the percentage of 1s in binary being >= .99. Finding the difference in entropy is relatively straight forward. I added in a print to my script for these two problems and printed the number of 1s and 0s when running on port 8001 and 8002. I ran the scripts a couple times each and found that there are less 1s in entropy 1 than in entropy 2 in random vs nonrandom numbers. For example, the average number of 1’s to 0’s in random vs nonrandom numbers is about 75% for entropy1. However, it is about 85% in entropy2. Looking at the two problems source code, I see that they are getting their numbers from different libraries ‘alteredrandom1’ and ‘alteredrandom2’. These must be the cause of producing numbers that are closer to the threshold of .99 in entropy2. Therefore, entropy2 is a more difficult challenge because the threshold for determining entropy (1’s to 0’s) must be exact. The numbers, on average, are closer to one another (as far as bit count) in entropy2, so it is harder to determine the entropy of the numbers. Therefore the difference in entropy between 1 and 2 is about 10% or .1 (85%-75%).

4g Substitution Cipher:
10 points

Problem:
For the ciphertext:

pkcycyszvfsqdovdosczpvfpwcpksxggnabvhevzli

Questions to answer:

What is the most frequent letter in the ciphertext?

What plaintext do you get from blindly mapping the most frequent ciphertext letters to the corresponding english letters (i.e. most frequent letter -> 'e', second most frequent -> 't', etc)?

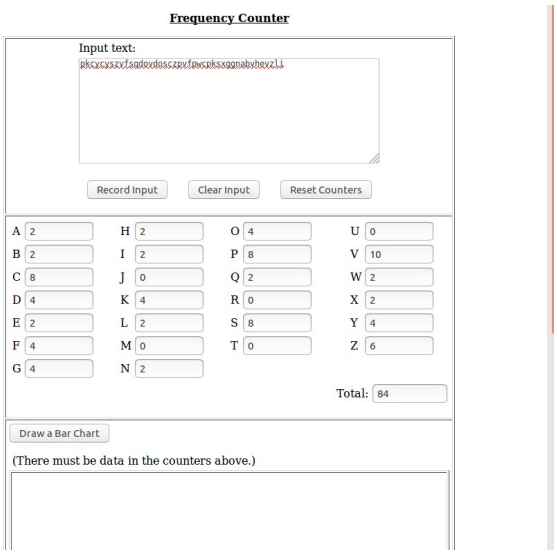
Now find the flag from:

NOTE: This question has been changed to make it a little more representative for frequency analysis. The server now uses 100 randomly generated words instead of 20, and has fewer rounds.

nc crypto.n0l3ptr.com 8004

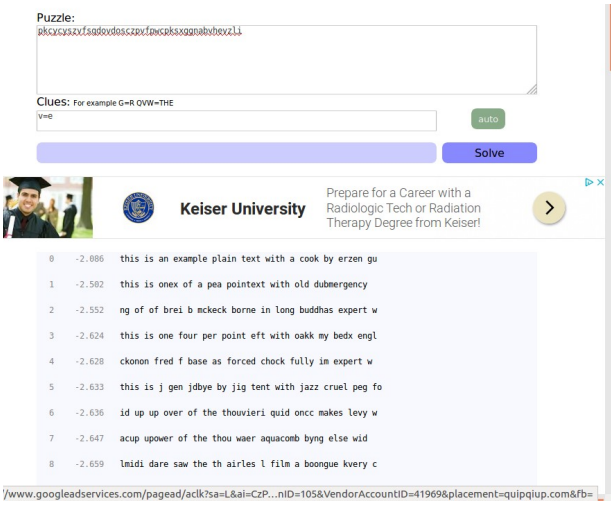
Solution:
What is the most frequent letter in the ciphertext?

Since the cipher text is actually 1 byte per character, I can use an online tool to count frequency of letters in a string. I used https://www.mtholyoke.edu/courses/quenell/s2003/ma139/js/count.html and input the string and found v to be the most frequent letter.

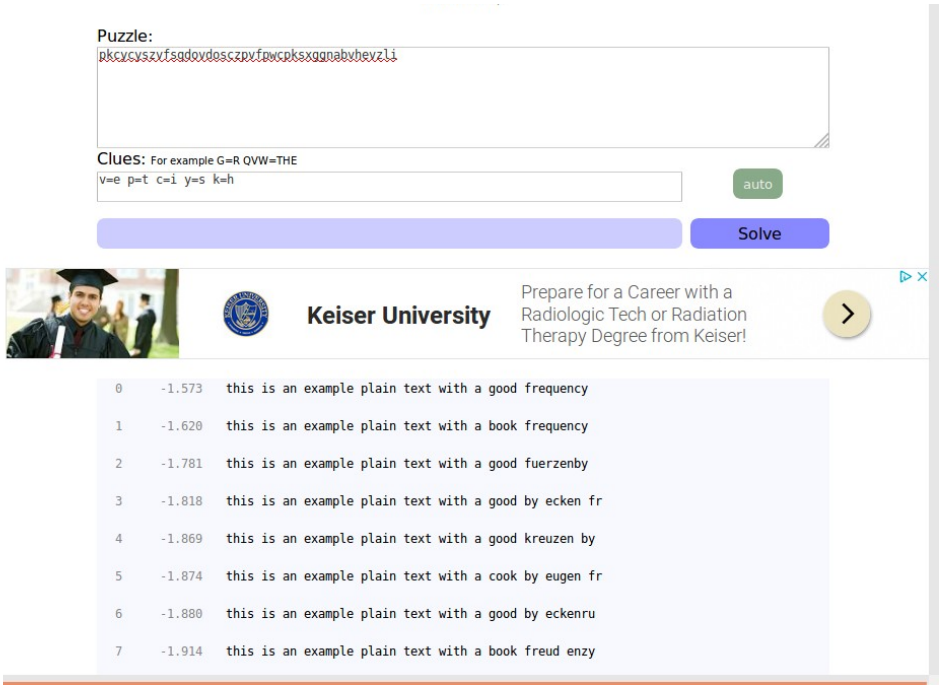


What plaintext do you get from blindly mapping the most frequent ciphertext letters to the corresponding english letters (i.e. most frequent letter -> 'e', second most frequent -> 't', etc)?

Since I can't directly map the frequency counter character to the ciphertext character because many ciphertext characters have the same frequency I will have to use an online tool. Using a substitution cipher cracker, found at https://quipqiup.com/, I cracked the plaintext. I knew the string was encrypted by matching letter occurrences of the text to the most frequent letters found in the frequency chart and swapping the letters. Therefore v is replaced with e. I put in the hint that v=e and found the following.



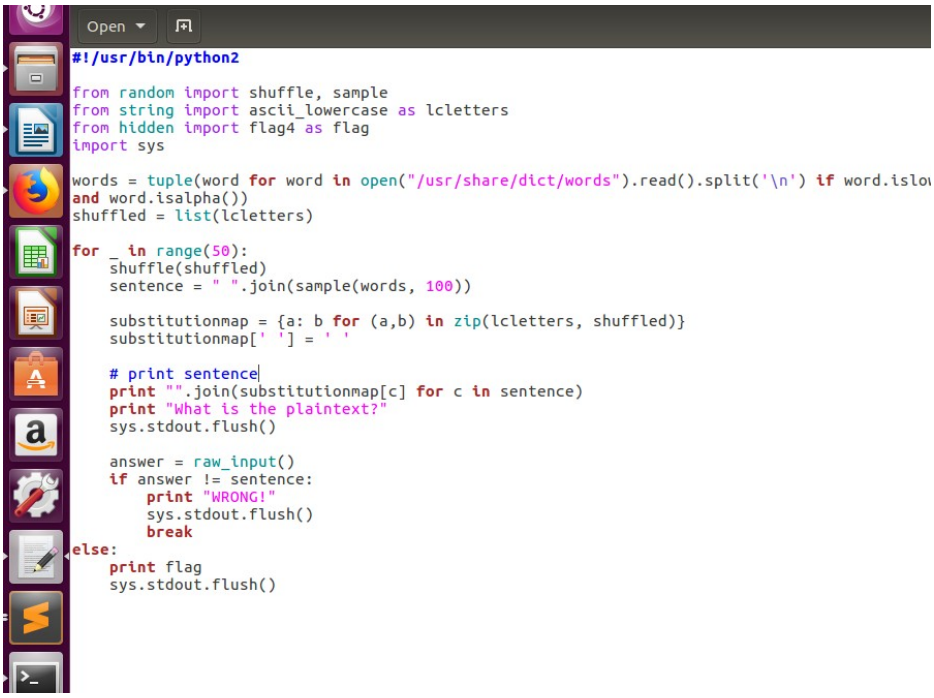
"This is an example" seems like plaintext. So I try mapping the following p=t c=i y=s k=h so that the first four letters would be 'this'. Inputting that into quibquib I find the answer.



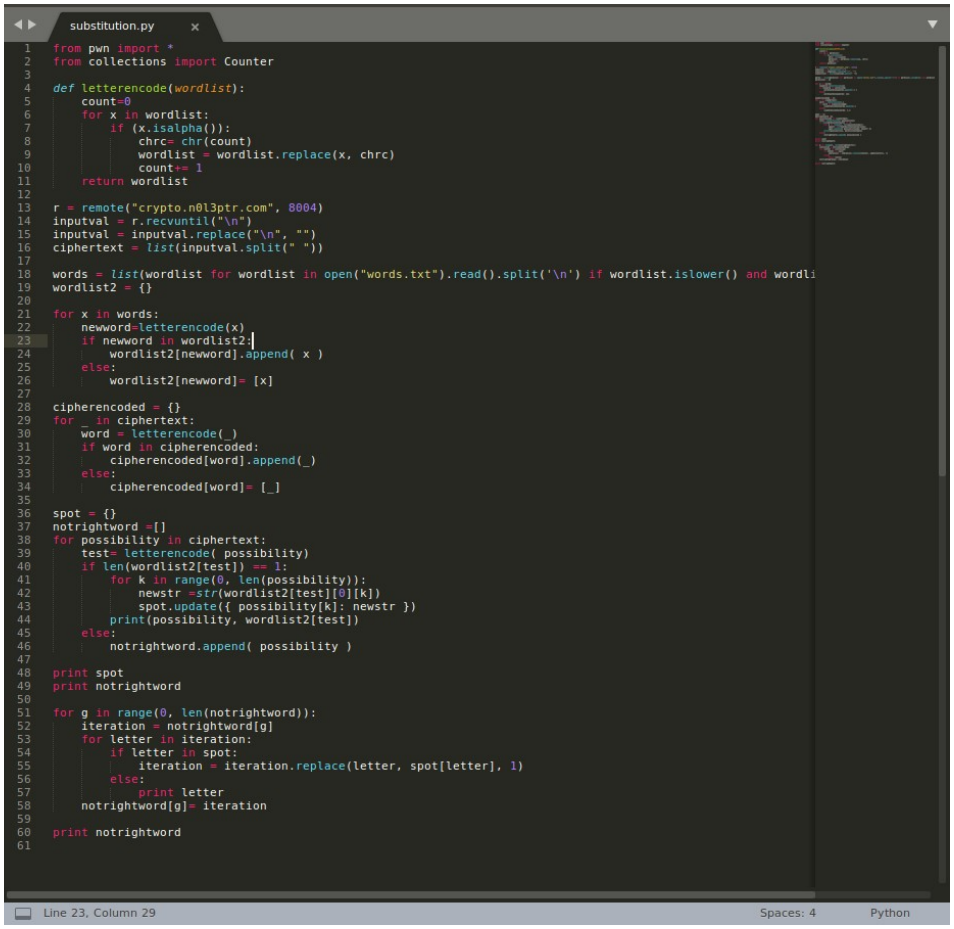
Plaintext = “thisisanexampleplaintextwithgoodfrequency”

Graduate portion:

We start by analyzing the provided source code ‘substitution.py’



Here we can see that after importing libraries, the code on the server generates a list of lowercase words from a file words on the server. Then in a loop that lasts 50 iterations, keys are created using shuffle and a random shuffling of lowercase letters is used as as the substitution for the substitution cipher. The original sentence is made to be 100 words from the words document. The program then prints the encoded text and asks for the plaintext 50 times. The flag is printed after providing the correct plaintext 50 times and isn’t printed if the wrong input is provided. This is confirmed through running the program. To solve this program I wrote a script ‘substitution.py’



After importing libraries and connecting to the server with pwntools, I parse the input and receive the encoded text with inputval, strip newlines, and split the ciphertext into words. I stole the words code from the provided script where Mitch receives words from ‘words.txt’. I have a words.txt file in my CWD that matches the one on Mitch uses for plaintext. The words.txt file was created through redirecting input from the server where the words file is stored. I then loop through the words, and use my function wordencode to create a byte pattern of matching words. This helps match ciphertext to encodings. This provides words to direct mappings. This can be used to get the substitution key for most of the letters as long as the word doesn’t have the same encoding as another. I create wordlist of these byte patterns and do the same thing with a variable ciphertext. I then loop through the ciphertext word bytes and check for direct mappings. If there is only one element stored in that index of byte words, the characters of the randomized word are mapped to the entries in the matching list index. The key to lowercase letter mappings are stored in the spot list. When direct mapping doesn’t work, the words are stored in a separate list. The correct mapping of words that match through direct mapping and the lowercase letter that maps to the key are printed to the program (ciphertext to plaintext mapping). I also print the words with multiple mappings. I then attempt to brute force the last keys that can’t be retrieved through direct mapping. I find out which letter is not being mapped but still can’t figure out how to find the key.

5g Single-Bye XOR:

20 points

Problem:

This hex-encoded string is encrypted by XORing each character with a single random keybyte:

a5bab3a6b3b5a0b7b3a6a2beb3bbbca6b7aaa6a6bdbdb0b3b6bba6b6bdb7a1bca6bab3a4b7bfbda0b7b7a1

What is the most frequent letter in the ciphertext?
What keybyte do you get with by assuming this letter is 'e'?
What plaintext do you get using that entire keybyte?

Now find the flag from:

nc crypto.n0l3ptr.com 8005

Solution:

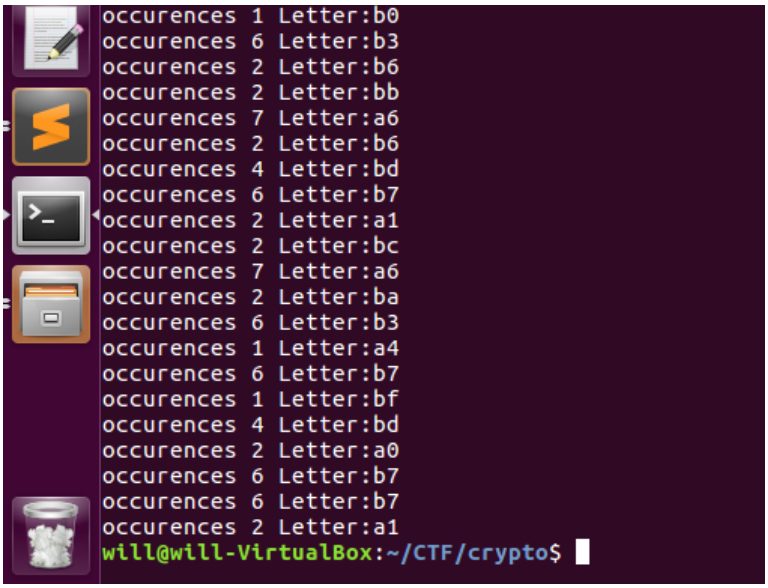
What is the most most frequent letter in the ciphertext?

‘a6’

Using the Python script below I found my answer

```
#!/usr/bin/env python
inp="a5bab3a6b3b5a0b7b3a6a2beb3bbbca6b7aaa6a6bdbdb0b3b6bba6b6bdb7a1bca6bab3a4b7bfbd
a0b7b7a1"
leng=len(inp)

inparr=[0]*(leng/2)
for i in range (0,(leng/2)):
  teststr=inp[i*2]+inp[i*2+1]
  for x in range(0,(leng/2)):
    if (teststr==(inp[x*2]+inp[x*2+1])):
      inparr[i]+=1
  print "occurences "+ str(inparr[i])+ " Letter:" + teststr
```

The given string is a hexstring. Since it takes two hex values to make an ASCII character we must check for repeated occurrences of consecutive bytes. This script does just that. It prints out the number of occurrences of every byte in the hexstring. The code is attached as ‘5.1.py’.



Here we can see ‘a6’ occurs 7 times which is the most.

What keybyte do you get with by assuming the letter is ‘e’?

Since we are assuming that the most frequent letter which is ‘a6’ is ‘e’ when it is decoded, we can find the key by reverse xoring. e^key = 0xa6. Therefore 0xa6 ^ 0x65 = the key. 0x65 is ‘e’. The keybyte is **195** (0x65 ^ 0xa5).

What plaintext do you get using that entire keybyte?

Using the following script we get the plaintext from the presumed key 195 by xoring every byte of the input with the key. Again the input is a hexstring so every two characters is a byte. The ‘char’ method is used to that the hex value can be printed.

```
#!/usr/bin/env python
inp="a5bab3a6b3b5a0b7b3a6a2beb3bbbca6b7aaa6a6bdbdb0b3b6bba6b6bdb7a1bca6bab3a4b7bfbda0b7b7a1"
plaintext=''
leng=len(inp)
for i in range(0,(leng/2)):
  substr=inp[i*2]+inp[i*2+1]
  substr=int(substr,16)
  substr=chr(substr ^ 195)
  plaintext+=substr
print plaintext
```

Output:
fypepvctpea}pxetiee~~spuxeu~tbeypgt|~cttb

The script is attached as ‘5.3.py’.

However, this plaintext is gibberish and not the flag. This goes to show that the most frequent letter occurrence is not always ‘e’, and we can not assume the key is based off of the most frequent letter. Through brute forcing the line ‘*substr=chr(substr ^ 195)’* and replacing 195 with 0-256 for the available ASCII characters that can be used as a key, we find that the key is actually 210.

The actual plaintext is whatagreatplaintexttoobaditdoesnthavemorees

Graduate portion:

First, I analyzed the provided source code.

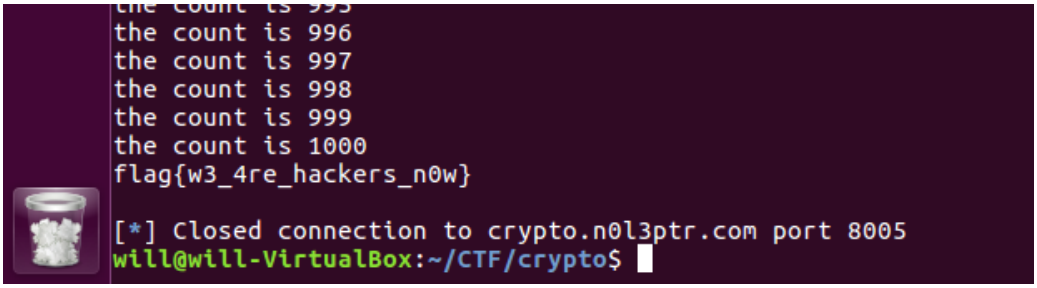
```python
#!/usr/bin/python2

from random import randint
from myutils import sxor #a function that xors two strings together
from random import shuffle, sample
from hidden import flag5 as flag
import sys

words = tuple(word for word in open("/usr/share/dict/words").read().split('\n') if wor

for _ in range(1000):
    keybyte = randint(0,255)
    sentence = "".join(sample(words, 20)) #no spaces this time!

    # print sentence
    print sxor(sentence, chr(keybyte)*len(sentence)).encode('hex')
    print "What is the plaintext?"
    sys.stdout.flush()

    answer = raw_input()
    if answer != sentence:
        print "WRONG!"
        sys.stdout.flush()
        break
else:
    print flag
    sys.stdout.flush()
```

Here we can see that dictionary words are read from a words file stored on the server. Then, in a loop that lasts 1000 iterations, a random key is generated and a sentence is created. In the loop, the sentence is xor'd with the key and the user is asked for the original sentence. If the incorrect sentence is sent to the program the program prints wrong and the loop ends. This is confirmed through running the program.

If the loop makes it through 1000 iterations a flag is printed. So, to get the flag, 1000 correct plaintext entries must be calculated. I did this by writing a script.

```python
from pwn import *
import binascii

r=remote("crypto.n0l3ptr.com", 8005)

i=0
while(i<1000):
    rawinput=r.recvuntil('\n')
    rawinput=rawinput.strip('\n')
    r.recvuntil('\n')

    unhex=binascii.unhexlify(rawinput)

    for key in range(0, 256):
        plaintext =""
        test = 0
        for x in unhex:
            hexval=ord(chr(key))
            testkey = hexval ^ ord(x)
            if (chr(testkey).isalpha() == False):
                test=1
            if (chr(testkey).islower() == False):
                test=1
            plaintext+= chr(testkey)


        if test == 0:
            r.sendline(plaintext)
            break

    i+=1
    print "the count is " + str(i)

output = r.recvuntil('\n')
print output
```

First, I import the correct libraries. I then connect to the server with pwntools. I then create a loop that lasts 1000 iterations. Inside the loop, I receive input until a newline which gets the encoded sentence. I then remove the newline from the string. After that, I receive until newline again so that I can remove the input prompt and send input to the server. This is all done using pwntools. I used the binascii library which I found details on at https://docs.python.org/2/library/binascii.html. Using the unhexifly method, I convert the provided encoded hex string to binary data. This data can be used to check for matching occurrences. I proceed to generate keys in the ASCII range. These keys match the keys used by the program on the server. Inside that loop, I iterate over the binary from unhexfily. I turn the key decimal number into a hex number and xor that with the binary data of the encoded string. I append the results as a character to a string that represents the generated plaintext. I then make sure that each character generated is a lowercase alphabetic character. This is because the sentences generated in the source code are lowercase alphabetic characters. If the generated plaintext string from xoring the binary data from the hex string contains all lowercase alphabetic characters, the plaintext is sent to the server and the iteration testing different keys breaks. This is done so that multiple plaintext strings are not being sent to the server. The loop for 1000 iterations then proceeds. At the end of the 1000 iteration loop loop we receive the flag. My script is attached as ‘xorsolve.py’.



flag{w3_4re_hackers_n0w}

Multi-Byte XOR

20 points

Problem:

Over the alphabet "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ", What is the expected value (https://nzmaths.co.nz/category/glossary/expected-value-discrete-random-variable) of each bit for all 8 bit positions? I.e. - bit 8 is 0 for all of these, so the expected value is 0.

Using pairwise XOR over that same alphabet, what is the expected value of each bit for all 8 bit positions? I.e. find the EV of all 8 bits of the bytes: 'a' XOR 'a', 'b' XOR 'a', ... 'Z' XOR 'Z'

5 point bonus: do both of these using the probability from the frequency chart from the slides and only the lowercase alphabet.

Consider the hex-encoded ciphertext:
851b255efc2f03aa820d2e54fd2a05ac97063b4ffa3108a3

If we guess that the key is 12 bytes, that means the first half of the ciphertext and the second half of the ciphertext were both encrypted with the same key. What is the entropy of C[:12] XOR C[12:]?

Now let's guess that the key is 8 bytes. What is the average entropy of C[:8] XOR C[8:16], C[8:16] XOR C[16:], and C[16:] XOR C[8:]?

Based on those results, do you think the key is length 12 or 8? Why?

Now decrypt this hex encoded plaintext by first finding the key length, then using the techniques we have learned this week:

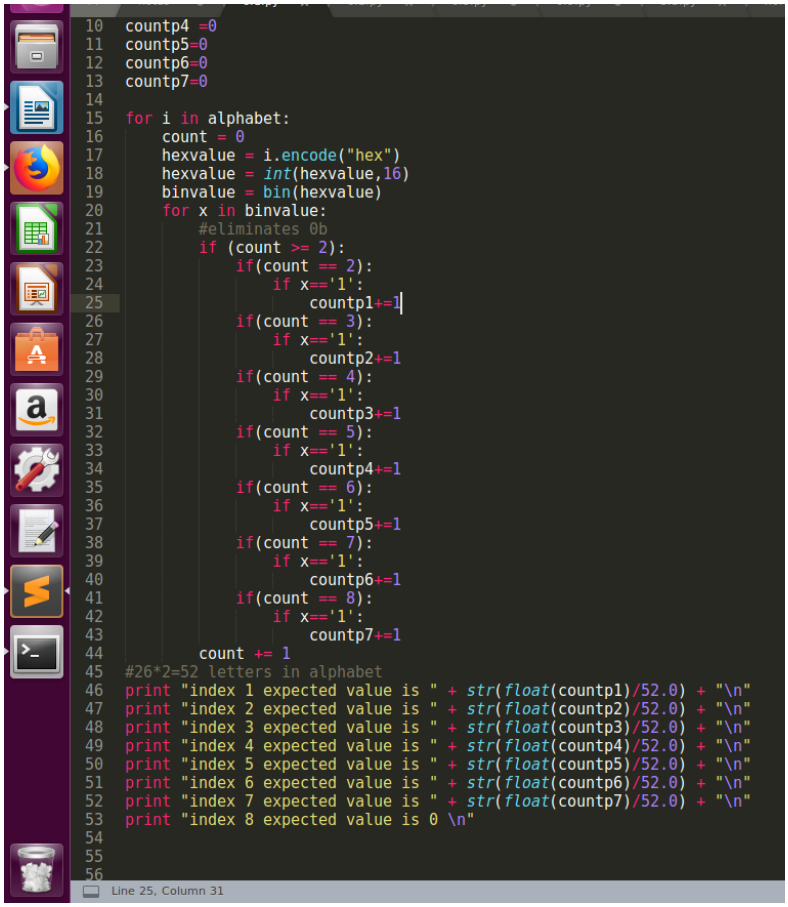
622cb216bdc0e21620be03f4c7f84221b71cbdddf95364b900eeddb14628ba0cf3ddf44e30fb2cbdc1f04021fb0…..

Solution:

Over the alphabet "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ", What

is the expected value (https://nzmaths.co.nz/category/glossary/expected-value-discrete-random-variable) of each bit for all 8 bit positions? I.e. - bit 8 is 0 for all of these, so the expected value is 0.

To solve this problem I wrote a script which is attached as ‘6.1.py’



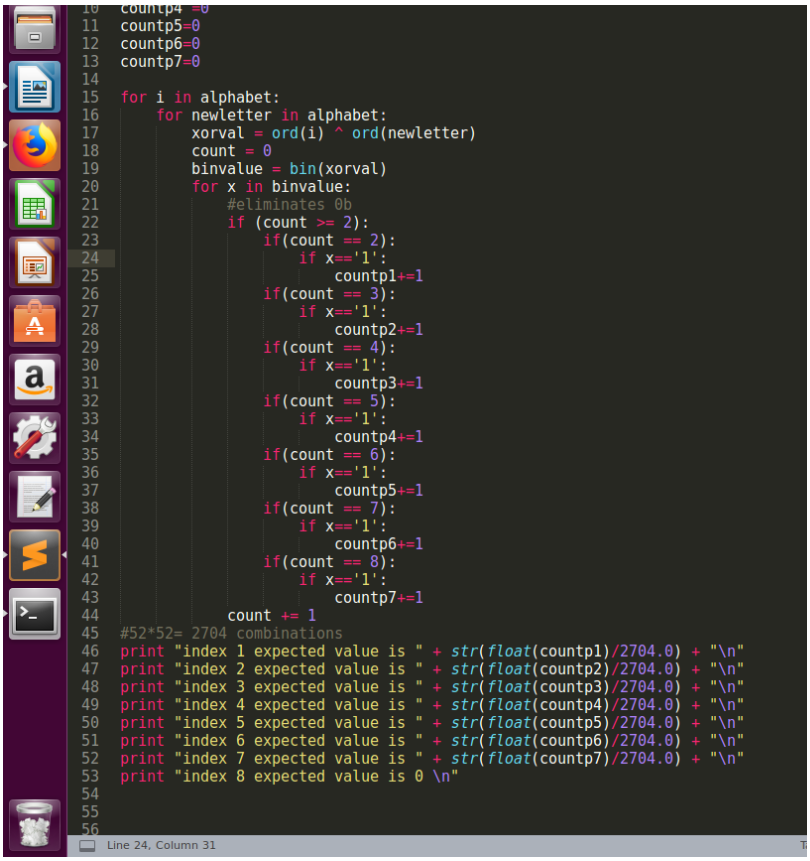
```
countp4 =0
countp5=0
countp6=0
countp7=0

for i in alphabet:
    count = 0
    hexvalue = i.encode("hex")
    hexvalue = int(hexvalue,16)
    binvalue = bin(hexvalue)
    for x in binvalue:
        #eliminates 0b
        if (count >= 2):
            if(count == 2):
                if x=='1':
                    countp1+=1
            if(count == 3):
                if x=='1':
                    countp2+=1
            if(count == 4):
                if x=='1':
                    countp3+=1
            if(count == 5):
                if x=='1':
                    countp4+=1
            if(count == 6):
                if x=='1':
                    countp5+=1
            if(count == 7):
                if x=='1':
                    countp6+=1
            if(count == 8):
                if x=='1':
                    countp7+=1
        count += 1
#26*2=52 letters in alphabet
print "index 1 expected value is " + str(float(countp1)/52.0) + "\n"
print "index 2 expected value is " + str(float(countp2)/52.0) + "\n"
print "index 3 expected value is " + str(float(countp3)/52.0) + "\n"
print "index 4 expected value is " + str(float(countp4)/52.0) + "\n"
print "index 5 expected value is " + str(float(countp5)/52.0) + "\n"
print "index 6 expected value is " + str(float(countp6)/52.0) + "\n"
print "index 7 expected value is " + str(float(countp7)/52.0) + "\n"
print "index 8 expected value is 0 \n"


Line 25, Column 31
```

I loop through the alphabet and encode every letter into hex and then into binary. I then loop through each character’s binary representation and count the number of 1s at each position. I do this because the number of 1’s compared to the total number of binary digits is how to calculate expected value. I start the counter at 2 because that ignores the 0b in 0b(100001). I then print out the count of 1’s for each bit index divided by 52. I divide it by 52 because there are 52 loop iterations (characters in the alphabet) where a bit could be a 0 or 1 at a given index. Number of 1 occurrences / 52 = expected value at an index for the alphabet.

```
will@will-VirtualBox:~/CTF/crypto$ subl 5.1.py
will@will-VirtualBox:~/CTF/crypto$ python 6.1.py
index 1 expected value is 1.0

index 2 expected value is 0.5

index 3 expected value is 0.423076923077

index 4 expected value is 0.423076923077

index 5 expected value is 0.461538461538

index 6 expected value is 0.5

index 7 expected value is 0.5

index 8 expected value is 0

will@will-VirtualBox:~/CTF/crypto$
```

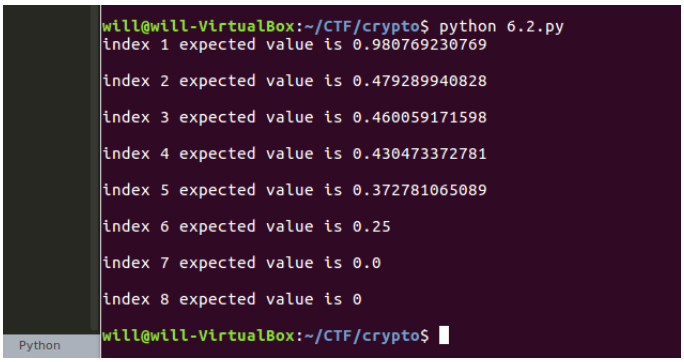
Using pairwise XOR over that same alphabet, what is the expected value of each bit for all 8 bit positions? I.e. find the EV of all 8 bits of the bytes: 'a' XOR 'a', 'b' XOR 'a', ... 'Z' XOR 'Z'

For this problem I wrote a script ‘6.2.py’



```
countp4 =0
countp5=0
countp6=0
countp7=0

for i in alphabet:
    for newletter in alphabet:
        xorval = ord(i) ^ ord(newletter)
        count = 0
        binvalue = bin(xorval)
        for x in binvalue:
            #eliminates 0b
            if (count >= 2):
                if(count == 2):
                    if x=='1':
                        countp1+=1
                if(count == 3):
                    if x=='1':
                        countp2+=1
                if(count == 4):
                    if x=='1':
                        countp3+=1
                if(count == 5):
                    if x=='1':
                        countp4+=1
                if(count == 6):
                    if x=='1':
                        countp5+=1
                if(count == 7):
                    if x=='1':
                        countp6+=1
                if(count == 8):
                    if x=='1':
                        countp7+=1
            count += 1
#52*52= 2704 combinations
print "index 1 expected value is " + str(float(countp1)/2704.0) + "\n"
print "index 2 expected value is " + str(float(countp2)/2704.0) + "\n"
print "index 3 expected value is " + str(float(countp3)/2704.0) + "\n"
print "index 4 expected value is " + str(float(countp4)/2704.0) + "\n"
print "index 5 expected value is " + str(float(countp5)/2704.0) + "\n"
print "index 6 expected value is " + str(float(countp6)/2704.0) + "\n"
print "index 7 expected value is " + str(float(countp7)/2704.0) + "\n"
print "index 8 expected value is 0 \n"


Line 24, Column 31
```

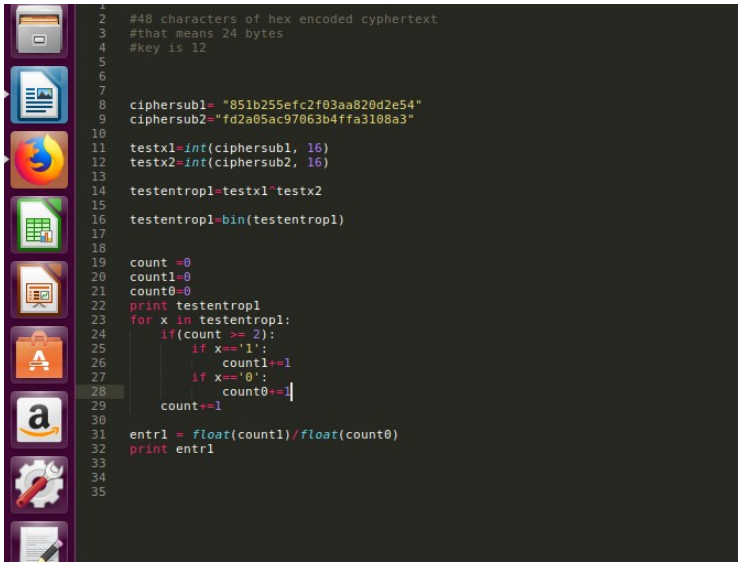
This script is essentially the same as ‘6.1.py’ However, I used a nested for loop to compare every letter against every other letter in the alphabet. Inside the nested loop, I xor the two character’s hex values. I then calculate the binary off of the result of that xor. The rest of the code is the same as ‘6.1.py’ except I calculate expected value using the number of 1 occurrences / 2704. I use 2704 because 2704 binary numbers are generated. The length of the alphabet is 52 characters and its in a nested loop. Therefore 52 *52 = 2704 iterations.



If we guess that the key is 12 bytes, that means the first half of the ciphertext and the second half of the ciphertext were both encrypted with the same key. What is the entropy of C[:12] XOR C[12:]?

I wrote a script to calculate entropy. The script is ‘6.3.py’

.



```
#48 characters of hex encoded cyphertext
#that means 24 bytes
#key is 12



ciphersub1= "851b255efc2f03aa820d2e54"
ciphersub2="fd2a05ac97063b4ffa3108a3"

testx1=int(ciphersub1, 16)
testx2=int(ciphersub2, 16)

testentropl=testx1^testx2

testentropl=bin(testentropl)


count =0
count1=0
count0=0
print testentropl
for x in testentropl:
    if(count >= 2):
        if x=='1':
            count1+=1
        if x=='0':
            count0+=1
    count+=1

entrl = float(count1)/float(count0)
print entrl
```

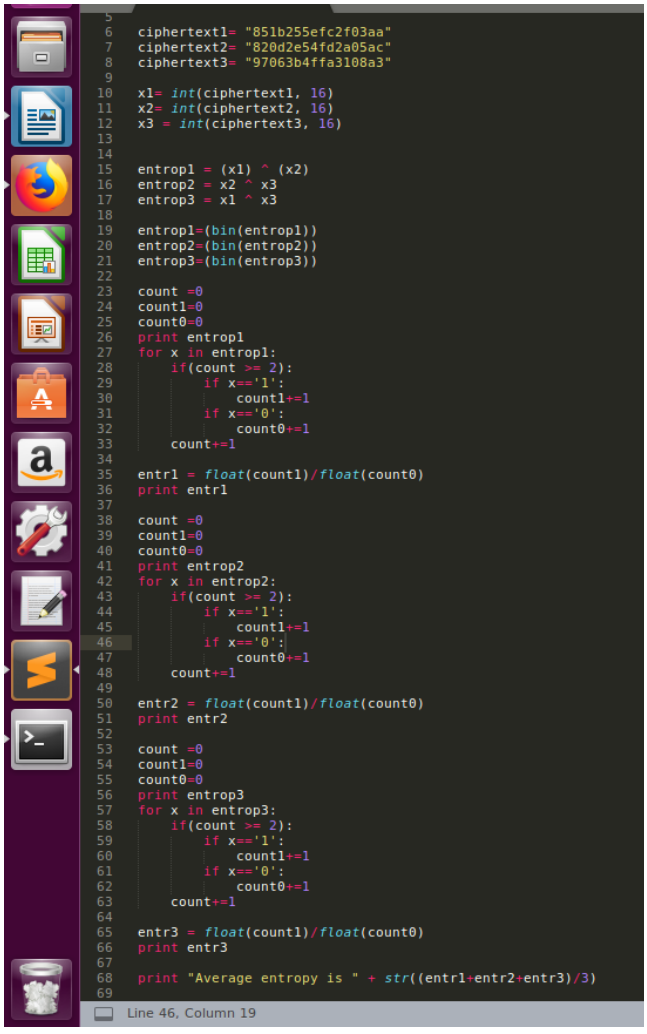
I convert each half of the string to hex, xor the two, and convert the result to binary. I then calculate the number of 0s and 1s and compare them. The entropy is the floating division of the count of 1s/ count of 0s.

```
0.979166666667
will@will-VirtualBox:~/CTF/crypto$ python 6.3.py
0b1111000001100010010000011110010011010110010100100111000111001010111100000111100001001101111011 1
0.979166666667
will@will-VirtualBox:~/CTF/crypto$
```

The entropy is .979.

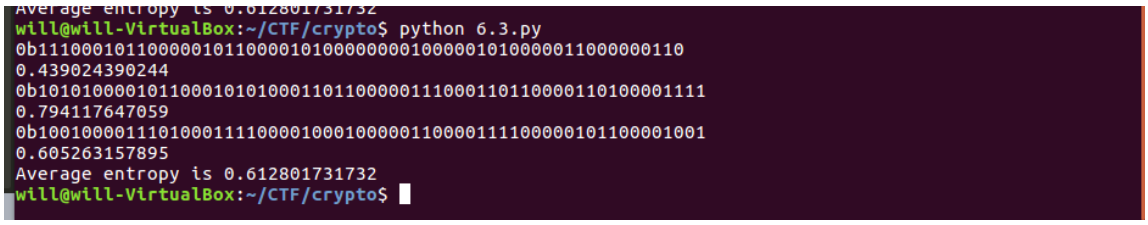
Now let's guess that the key is 8 bytes. What is the average entropy of C[:8] XOR C[8:16], C[8:16] XOR C[16:], and C[16:] XOR C[8:]?

I took the same approach for this problem as I did on the previous question. However, I calculated three entropies of smaller substrings of 16 characters. The code is attached as '6.4.py'



```
ciphertext1= "851b255efc2f03aa"
ciphertext2= "820d2e54fd2a05ac"
ciphertext3= "97063b4ffa3108a3"

x1= int(ciphertext1, 16)
x2= int(ciphertext2, 16)
x3 = int(ciphertext3, 16)


entrop1 = (x1) ^ (x2)
entrop2 = x2 ^ x3
entrop3 = x1 ^ x3

entrop1=(bin(entrop1))
entrop2=(bin(entrop2))
entrop3=(bin(entrop3))

count =0
count1=0
count0=0
print entrop1
for x in entrop1:
    if(count >= 2):
        if x=='1':
            count1+=1
        if x=='0':
            count0+=1
    count+=1

entr1 = float(count1)/float(count0)
print entr1

count =0
count1=0
count0=0
print entrop2
for x in entrop2:
    if(count >= 2):
        if x=='1':
            count1+=1
        if x=='0':
            count0+=1
    count+=1

entr2 = float(count1)/float(count0)
print entr2

count =0
count1=0
count0=0
print entrop3
for x in entrop3:
    if(count >= 2):
        if x=='1':
            count1+=1
        if x=='0':
            count0+=1
    count+=1

entr3 = float(count1)/float(count0)
print entr3

print "Average entropy is " + str((entr1+entr2+entr3)/3)

Line 46, Column 19
```

Here I just repeat the count and calculation for entropy for all three xor's, just like I did in the previous problem. I then print the average of their entropies. The entropy average is .613.



Based on those results, do you think the key is length 12 or 8?

The key is of length 12. 12 provides a higher entropy so the characters are more random.