Kunal Mukherjee
CS 6324
2/8/21
Dr. Chung Hwan Kim

Homework #1

Problem 1:

a> Confidentiality: it is a property that only lets the entities who are authorized to know the data can know that data.
Integrity: it is the property that prevents unauthorized parties from modifying the data. Also, only let the authorized entities modify in permitted ways and not give all access editing permission to anyone.
Availability: it is the property that says that the authorized entities should have access to the data at all times.

b> 1.1. Anthem data breach, when hacker got unauthorized access to customer's SSN, DOB, address. Confidentiality is breached.
1.2. Equifax data breach, when hackers got unauthorized access to customer's SSN, DOB, address. Confidentiality is breached.
1.3 when you are typing your bank's password and someone from behind looks over and sees your password. He sees your confidential information.

2.1. A hacker hijacks a banking session and changes the amount requested by the authorized user. Thus, violating the integrity of the session.
2.2. A hacker changes a package manager to embed its own malicious code. Thus, violating the integrity of the package manager. And when this package manager will be used to compile a code or do any data processing, the intended output will be modified, as well as contain a malicious payload due to using the malicious package manager.

3.1. Denial of service, when a hacker overloads all the server ports so that valid clients cannot connect to it or have access to it. Example, CloudFlare attack in 2014, when attackers exploited vulnerability in the Network Time Protocol (NTP) protocol. Since, authorized entities did not have access to the data. It violates availability.
3.2. Six Bank DDoS Attack in 2012, it was a distributed denial of service, where bots took down six banks, Bank of America, JPMorgan Chase, U.S. Bank, Citigroup, Wells Fargo, and PNC Bank. Therefore, authorized entities did not have access to the data so it violated availability.

c> 1. Session cookies- protection against cross-site scripting (XSS) attacks. Thus, protecting against data integrity. Session keys are vulnerable to man-in-the-middle attack. Session key hijack can be done by a man-in-the-middle who listens to the network in promiscuous mode and modify the cookie that is being sent to the server and client. So, the client will think the MITM is the server and will send it all the content to it and MITM will take the contents modify it however it wants and send it to the server. Then, do the reverse when it gets the reply from the server. So, client will think it is a valid response to its request and the server will think it is a valid request from a client.

2. Stack canaries- unauthorized stack modification by the current program or another program. It is used to protect the computer execution stack. Thus, providing data integrity. But stack canaries are vulnerable to brute force attacks and side-channel memory disclosure attacks.

Problem 2:

1. a) Let A = event that a double is rolled A.
   elements in the event A = (1,1), (2,2) … (6,6)
   size of all number of possible elements in A = |A| = 6
   Total number of possible elements = 6x6 = 36.
   So, there is 6 ways to roll double. So, probability of rolling a double is 6/36 = 1/6.

   b) Let A = event that roll result is a sum of 6 or less
   A = {(1,1), (1,2), (1,3), (1,4), (1,5), (2,1), (2,2), (2,3), (2,4), (3,1), (3,2), (3,3), (4,1), (4,2), (5,1)}, |A| = 15. p(A) = 15/36.
   B = event that a double is rolled.
   |B| = 6 (from 1a).
   AB = The event that a double is rolled and the sum of 6 or less = {(1,1), (2,2), (3,3)}, |AB| = 3. P(A^B) = 3/36.
   So, p(B|A) = p(A^B)/p(A) = (3/36) / (15/36) = 3/15 = 1/5

   c) Let A= event that the larger of the two die is at least 4
   A = {(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6),(4,1)…(4,6),(5,1)…(5,6),(6,1)…(6,6)}.
   |A| = 27. p(A) = 27/36.

   d) F = event that two-die land on different numbers
   = all possible elements - event that a double is rolled A = 36 – 6 = 30.
   P(F) = 30/36.
   E = event at least one die is 1.
   FE = event at least one die is 1 and two die land on different numbers = {(1,2) …(1,6), (2,1),(3,1),(4,1),(5,1),(6,1)} = 10. p(FE) = 10/36.
   So, p(E|F) = p(E^F)/p(F) = (10/36) / (30/36). = 10/30 = 1/3.

   e) X = first die is an odd no.
   Y = rolled sum is an even no.
   X = {(1,1)…(1,6),(3,1)…(3,6), (5,1)…(5,6)}. |X| = 18. p(X) = 18/36 = ½.
   Y = {(1,1),(1,3),(1,5), (2,2),(2,4),(2,6), (3,1),(3,3),(3,5), (4,2),(4,4),(4,6), (5,1),(5,3),(5,5), (6,2),(6,4),(6,6)}. |Y| = 18. p(Y) = 18/36 = ½ .
   X^Y = {(1,1), (1,3),(1,5), (3,1),(3,3),(3,5), (5,1),(5,3),(5,5)}. |X^Y| = 9. p(X^Y) = 9/36.
   p(X|Y) = p(X^Y)/p(Y) = (9/36) / (18/36) = ½

Since, P(X) = p(X|Y). So, X and Y are independent.

f) X = first die is a multiple of 2.
Y = rolled sum is a multiple of 2.
X = {(2,1)…(2,6),(4,1)…(4,6), (4,1)…(4,6)}. |X| = 18. p(X) = 18/36 = ½.
Y = {(1,1),(1,3),(1,5), (2,2),(2,4),(2,6), (3,1),(3,3),(3,5), (4,2),(4,4),(4,6), (5,1),(5,3),(5,5), (6,2),(6,4),(6,6)}. |Y| = 18. p(Y) = 18/36 = ½ .
X^Y = {(2,2),(2,4),(2,6), (4,2),(4,4),(4,6), (6,2),(6,4),(6,6)}. |X^Y| = 9. p(X^Y) = 9/36.
p(X|Y) = p(X^Y)/p(Y) = (9/36) / (18/36) = ½
Since, P(X) = p(X|Y). So, X and Y are independent.

g) X = first die is a multiple of 3.
Y = rolled sum is a multiple of 2.
X = {(3,1)…(3,6),(6,1)…(6,6)}. |X| = 12. p(X) = 12/36 = 1/3.
Y = {(1,1),(1,3),(1,5), (2,2),(2,4),(2,6), (3,1),(3,3),(3,5), (4,2),(4,4),(4,6), (5,1),(5,3),(5,5), (6,2),(6,4),(6,6)}. |Y| = 18. p(Y) = 18/36 = ½ .
X^Y ={(3,1),(3,3),(3,5),(6,2),(6,4),(6,6)}. |X^Y| = 6. p(X^Y) = 6/36.
p(X|Y) = p(X^Y)/p(Y) = (6/36) / (18/36) = 1/3.
p(Y|X) = p(X^Y)/p(X) = (6/36) / (12/36) = 1/2.
Since, P(X) = p(X|Y). So, X and Y are independent.

2. X – event a student pass midterm. p(X) = 0.75.
Y – event a student pass final. p(Y) = 0.8.
p(Y|X) = 0.9
p(Y|X) = p (X^Y)/p(X)
=> p(X^Y) = p(X)p(Y|X) = 0.75 x 0.9 = 0.675.
p(Y|!X) = p(Y^!X)/p(!X) = p(Y) – p(X^Y)/1-p(X) = 0.8-0.675/1-0.75 = ½
p(!X|Y) = p(Y^!X)/p(Y) = p(Y) – p(X^Y)/p(Y) = 0.8-0.675/0.8 = 5/32
Since, p(Y) = 0.75 != 0.9 = p(Y|X), so X and Y is not independent.

3. a. M – applicant interviewed are married
F – applicants at least 5 years teaching experience.
p(M) = 30+18/78 = 48/78
p(F) = 18+12/78 = 30/78
p(M^F) = 18/78
p(M|F) = p(M^F)/p(F) = (18/78) / (30/78) = 18/30.
P(F|M) = p(M^F)/p(M) = (18/78) / (48/78) = 18/48.
Since, p(F|M) != P(F), so F and M are not independent.

b. U – applicant interviewed are single
V – applicants less than 5 years teaching experience.
Also given, at least 5 year exp has twice the chance of an application than with less than 5 year experience.

M	F	p(M,F)
0	0	6/48
0	1	8/30
1	0	10/48
1	1	12/30

p(U) = p(!M) = 30 + 64 / 240 = 94/240
p(V) = p(!F) = 1 – 2/3 = 1/3

p(U^V) = 6/48
p(U|V) = p(U^V)/p(V) = 6/48 / 1/3 = 3/8
p(V|U) = p(U^V)/p(U) = 6/48 / 94/240 = 6 * 240 / 48 * 94
Since, p(U|V) != P(U), so U and V are not independent.

4. a. S – person has disease
T – test result +ve

p(T|S) = .95
p(!T|!S)=.95
p(S) = 0.001
p(!S) = 1 – p(S) = 1-0.001 = 0.999
p(!T|S) = 1 – p(T|S) = 1 – 0.95 = 0.05
p(T) = p(T^S) + p(T^!S) = p(T|S)p(S) + p(!T|S}P(S)
= (.95)(0.001)+(0.05)(0.999) = 0.0509
p(T^S) = p(T|S)p(S) = 0.95x0.001 = 0.00095
p(S|T) = p(T^S)/p(T) = 0.00095/0.0509 = 0.0186

b. p(T|S) = .998
p(!T|S) = 1 – p(T|S) = 1 – 0.998 = 0.002
p(T) = p(T^S) + p(T^!S) = p(T|S)p(S) + p(!T|S}P(S)
= (.998)(0.001)+(0.002)(0.999) = 0.0509
p(T^S) = p(T|S)p(S) = 0.998x0.001 = 0.000998
p(S|T) = p(T|S)*p(S) / p(T) = 0.998 * 0.001 / 0.0509 = 499/25450

c. p(!T|!S)=.998
p(T|!S) = 1 – p(T|S) = 1 – 0.95 = 0.05
p(T) = p(T^S) + p(T^!S) = p(T|S)p(S) + p(T|!S}P(!S)
= (.95)(0.001)+(0.05)(0.999) = 0.0509
p(T^S) = p(T|S)p(S) = 0.95x0.001 = 0.00095
p(S|T) = p(T^S)/p(T) = 0.00095/0.0509 = 0.0186

Problem 3:

a> 1. ciphertext-only attack – the adversary only knows one or some cipher text. But they have no knowledge of the corresponding plain text.
2. known-plaintext attack - adversary knows only one or some pairs of ciphertext and corresponding plaintext. But the adversary doesn't get to choose, they have at least one pairs, and may have many pairs.
3. chosen plaintext attack - A chosen plaintext attack is the same thing as known-plaintext attack except the adversary get to choose the plaintext as well as the corresponding ciphertext. Therefore, it can be useful in constructing a key. Thus, the adversary choses advantageous plaintext that will be encrypted.

b> 1. Attacker can use frequency analysis to break substitution cipher under a ciphertext-only attack. Cryptanalysis can be used, as the substitution cipher preserves the inherent language properties such as frequency of occurrence and it can be exploited.
2. The simplest way to break under known-plaintext attack will be to reverse of the operation that is being done to encrypt. For Vigenere cipher, you can just subtract the ciphertext from the plaintext to get the key and for OTP: the key will be plaintext xor ciphertext.

c> Assumption: the message sent from the laptop is the PT, and the message coming from the wireless access point is the CT. While in reality, laptop never sends the PT, it is encapsulated inside SSL/TLS security. But for this problem we are ignoring it.

1. Attacker can put his computer in promiscuous mode and sniff the network packets. Then the attacker can send some of their packets in the network so that it gets encrypted via the access point, then he can sniff them out. But he will have to choose his cyphertext vs the entire network traffic. Thus, he cannot pick the cipher text corresponding to the plaintext directly. So, the hacker knows some pair of ciphertext and corresponding plaintext.

2. The attacker to employ chosen-plain text attack, they have to be the man-in-the-middle. So, they will first have to attack the routing table of the PC and poison it to replace its address where the wireless access point's address was. So, the PC will think it is connecting to the wireless point but instead will connect to the attacker. And, the attacker will be the man-in-the-middle. So, the now, the attacker can listen to the packets from the PC in plaintext and send it to the wireless point and get the cipher text back. Now, the attacker has both the cipher text and the plain text, and he has the power to choose, so it is chosen plain text attack.

Problem 4:

a> Double encryption does not increase the security for double Vigenere cipher against a ciphertext only attack. The issue with Vigenere cipher is that it does not mask the frequency features or the statistical information of the underlying language. So, we can employ frequency analysis on the cipher text to learn about the key.

For our example, I used a variant of Kasiski attack. Here, I am assuming we will use the English language as the underlying native language. For English we know that 'e' will the highest frequent letter that will be used and it will be around 13% based on cryptographic letter frequency handout[1,2]. So, to analyze the cipher text I will select all the ciphertext that appear in the position multiple of LCM(l, l+1), e.g.1, LCM(l,l+1), 2*LCM (l, l+1), …,x*LCM (l, l+1), where x goes from 1 to inf. until I find the ciphertext letter that consists of 13% of the selected ciphertext. Then, we will know that cipher text maps to e. We are looking for 13% because 13% of the selected letters will be made of plaintext of letter 'e'. Then, we will go onto analyze to find, 't' which has 9.1% and then 'a' 8.2%, and so on.

b> In this scenario, we have a range of l. So, our k1 can have length, 10 <= len(k1) <= 19 and k2 can have length, 11 <= len(k2) <= 20. But, in this case our job becomes a lot easier because we are given the corresponding plaintext. So, we know that this relationship exists:
C' = P + k1 (mod 26)
C = C' + k2 (mod 26) => C = P + (k2 + k1) (mod 26)

P' = C' – k2 (mod 26)
P = P' – k1 (mod 26) => P = (C' – k2) – k1 (mod 26) or P = C' – (k2 + k1) mod 26.

We can set k'' = k2 + k1, so, our equations become:
C = P + k'' (mod 26) and P = C – k'' (mod 26). So, k'' = C – P (mod 26).

The possible keys are in the length of the k'' from 10 to 20 as that is the range of length of l and l+1. So, for a cipher text of length 50, we know k'' can be anywhere from starting at index 0 to 30. Then we split the ciphertext into, k''(length of key), columns consider each column as a single string encrypted using Caesar cipher. Then, we can again perform a frequency analysis, but on each string. And using the frequency table and our knowledge of frequency properties of the language and we will the map the key to letter. We have to do it at most 20 times, because our key can have length starting from 10 and can be as large as 20.

While the above method, will be my primary method of choice, there is another way of attacking this using Kasiski examination. We know it has been encrypted twice, so we can create the Kasiski table twice, to turn this two-poly sub cipher into a mono-substitution cipher. For example, for every possible k in l < k < l-1, where 10 <= l <= 19 we will line up the ciphertext in n columns, where n is the length of the keyword. Then treat each column as a mono-alphabetic substitution cipher. And do it twice. Then finally do the frequency analysis on the resulting columns of the second table.

[1] https://en.wikipedia.org/wiki/Letter_frequency
[2] https://www3.nd.edu/~busiforc/handouts/cryptography/letterfrequencies.html

Problem 5:

a. Problem5.py added to the submission. It contains the whole implementation. I am adding the core part of the code, that contains the creation of the y and the encryption and decryption, as well as how to get the original plaintext.

```python
def makeY(x, option):
    _temp_y = x
    y = []
    for i, xi in enumerate(x):
        if 0 <= i <= 16:
            y.append(xi)
        else:
            if option == "encrypt":
                _y = (ord(x[i]) + ord(y[i - 17])) % 26
            else:
                _y = (ord(x[i]) - ord(_temp_y[i - 17])) % 26

            _y += ord('A')
            y.append(chr(_y))
    y = "".join(y)

    return y
```

```python
def encrypt(PT, key):

    cipher_text = []
    for i, pt in enumerate(PT):
        x = (ord(pt) + ord(key[i])) % 26
        x += ord('A')
        cipher_text.append(chr(x))

    return "".join(cipher_text)


def decrypt(CT, key):
    orig_text = []
    for i, ct in enumerate(CT):
        x = (ord(ct) - ord(key[i])) % 26
        x += ord('A')
        orig_text.append(chr(x))

    y_dec = "".join(orig_text)
    # print("_y:  ", y_dec)

    plaintext_dec = makeY(y_dec, "decrypt")
    # print("_pt: ", plaintext_dec)
    # print("pt:  ", plaintext_dec[RAND_STR_LEN:])

    return plaintext_dec[RAND_STR_LEN:]
```

```
def main():

    # answer 5a,b
    plaintext1 = "ABCDEFGGHIJKLMNOPQRSTUVWXYZ"
    print("pt:    {}".format(plaintext1))

    KEY = [chr(int(random.uniform(0, 26)) + ord('A')) for _ in range(0,
len(plaintext1)+RAND_STR_LEN)]
    print("key:   {}".format("".join(KEY)))

    print("")

    for i in range(3):
        rand_str = getRandomStr(RAND_STR_LEN)
        print("rnd str (len 17): {}".format(rand_str))

        x = rand_str + plaintext1
        print("x: {}".format(x))

        y = makeY(x, "encrypt")
        print("y: {}".format(y))

        ct = encrypt(y, KEY)
        print("ct:   ", ct)

        plaintext_dec = decrypt(ct, KEY)
        print("pt_dec: ", plaintext_dec)

        print("")
```

b.

```
pt:   ABCDEFGHIJKLMNOPQRSTUVWXYZ
key:   AXJRFJAVZBRSQEXAOZNHBTWLNHLWEYWNQTZUVRPVCBLF


rnd str (len 17): XDOPHYEKFDGGTMHKY
x: XDOPHYEKFDGGTMHKYABCDEFGHIJKLMNOPQRSTUVWXYZ
y: XDOPHYEKFDGGTMHKYXEQSLDKRNMQRFZVZOOWJMGZHPM
ct:    XAXGMHEFEEXYJQEKMWRXTEZVEUXMVDVIPHNQEDVUJQX
pt_dec:  ABCDEFGHIJKLMNOPQRSTUVWXYZ


rnd str (len 17): BLOAEWYJOZCEIRSTN
x: BLOAEWYJOZCEIRSTNABCDEFGHIJKLMNOPQRSTUVWXYZ
y: BLOAEWYJOZCEIRSTNBMQDIBEQWIMPUEGIDSEJXDXBOV
ct:    BIXRJFYENATWYVPTBAZXEBXPDDTITSATYWRYEOSSDPG
pt_dec:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
rnd str (len 17): IZFPNJYJXCDKUNBLC
x: IZFPNJYJXCDKUNBLCABCDEFGHIJKLMNOPQRSTUVWXYZ
y: IZFPNJYJXCDKUNBLCIAHSROEQFLNVGAPASZSAMMKBOE
ct:   IWOGSSYEWDUCKRYLQHNOTKKPDMWJZEWCQLYMVDBFDPP
pt_dec:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

c. Using Shannon's maxim that "The enemy knows the system". I am reverse engineer the cipher text to get the key, then use the key to decrypt the cipher text c2.

For a given plaintext ciphertext, we will use this algo:

i> Algorithm:

```
recoverKey(M1, C1, C2):
   i = 0 // index var
   new-key = [] //array to hold new generated key
   for each plaintext in M1 do:
         if i <17: // we are dealing with random string
               new-key.append(C1[i] – M1[i] % 26)
         else:
               new-key.append(C1[i] = M1[(i – 17) % 17] % 26)
         i++
   return new-key
```

ii> Output:

```
org key   ZMFMZANWEZHDQNWGHITLDQVFFBSOHOASOHBGYEGQXESQKEIZPY

rnd_str1 (len 17): GEVOAIWSESKTHXHSC
pt1: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
M1(rnd_str1+pt1):
GEVOAIWSESKTHXHSCAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
y1 GEVOAIWSESKTHXHSCGEVOAIWSESKTHXHSCGEVOAIWSESKTHXHS
CT1:  FQAAZIJOIRRWXKDYJOXGRQDBXFKYAVXZGJHKTSGYTWWIUXPWWQ

rnd_str2 (len 17): MZBTDTLSQPRTSDLNQ
pt2: ABCDEFGHIJKLMNOPQRSTUVWXYZ
M2(rnd_str2+pt2): MZBTDTLSQPRTSDLNQABCDEFGHIJKLMNOPQRSTUVWXYZ
y2 MZBTDTLSQPRTSDLNQMADWHYRZYYBEEQZCGDSWQCUOXX
CT2:  LLGFCTYOUOYWIQHTXUTOZXTWEZQPLSQRQNEYUUIKLBP

CT1:   FQAAZIJOIRRWXKDYJOXGRQDBXFKYAVXZGJHKTSGYTWWIUXPWWQ
M1:    GEVOAIWSESKTHXHSCAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-
KEY:   ZMFMZANWEZHDQNWGHITLDQVFFBSOHOASOHBGYEGQXESQKEIZPY
```

```
y2:  MZBTDTLSQPRTSDLNQMADWHYRZYYBEEQZCGDSWQCUOXX
M2:  MZBTDTLSQPRTSDLNQABCDEFGHIJKLMNOPQRSTUVWXYZ
pt2:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

iii> Python implementation

```
def recoverKEY(M1, CT, CT2):
    print(" ")

    new_key = []

    for i, (xi, ci) in enumerate(zip(M1, CT)):
        if i < RAND_STR_LEN:
            g = (ord(ci) - ord(M1[i])) % 26
            g += ord('A')
            new_key.append(chr(g))

        else:
            g = (ord(ci) - ord(M1[(i - RAND_STR_LEN) % RAND_STR_LEN])) %
26
            g += ord('A')
            new_key.append(chr(g))

    new_key = "".join(new_key)
```

d. We will attack the cipher in ciphertext only attack by first trying to separate out 17 random character and the actual message. We know that the first 17 characters of x are random, and they will be encrypted also. So, the resulting ciphertext will have some random characters in-between the normal encryption and that is going to make the detection of the length of key hard to make. Thus, making it hard to recreate the key.

   If we have the cipher text longer than key, then we can use Kasiski tests and if the ciphertext is large and the underlying message gets repeated letters due to birthday paradox, we can use the distance between the repetition and deduce the key length. Once, we get the key length, if we see it is small we can brute-force through it. But, if the key length is large, we have to do frequency analysis. For example, we split the ciphertext into, k(length of key), columns consider each column as a single string encrypted using Caesar cipher. Then, we can again perform a frequency analysis, but on each string. And using the frequency table and our knowledge of frequency properties we can manage to map the key to letter.

Problem 6:

a> Take a look at “problem6.py”, so see the code behind the generation of the table and answered a and b.

Table[PT][K] = CT

table for PT x K:
```
 [[ 1.  2.  3.  4.  5.  6.]
 [ 2.  4.  6.  8. 10.  0.]
 [ 3.  6.  9.  0.  3.  6.]
 [ 4.  8.  0.  4.  8.  0.]
 [ 5. 10.  3.  8.  1.  6.]
 [ 6.  0.  6.  0.  6.  0.]]
```

table for PT=5,6
```
 [[ 5. 10.  3.  8.  1.  6.]
 [ 6.  0.  6.  0.  6.  0.]]
```

```
CT:1.0 PT:[5] K:[5]
CT:3.0 PT:[5] K:[3]
CT:5.0 PT:[5] K:[1]
CT:8.0 PT:[5] K:[4]
CT:10.0 PT:[5] K:[2]
```

For CT [1.0, 3.0, 5.0, 8.0, 10.0] we can guess the value of M or PT = 5. Because, for those cipher texts, they have only one corresponding plaintext.

b> Pr [PT=p| CT=i] for each i in [1,…,12] and each p in [1,…,6]

Pr [PT=1 | CT=1] = 0.03 / 0.06 = 0.5
Pr [PT=1 | CT=2] = 0.03 / 0.06 = 0.5
Pr [PT=1 | CT=3] = 0.03 / 0.11 = 0.25
Pr [PT=1 | CT=4] = 0.03 / 0.11 = 0.25
Pr [PT=1 | CT=5] = 0.03 / 0.06 = 0.5
Pr [PT=1 | CT=6] = 0.03 / 0.22 = 0.125
Pr [PT=1 | CT=7] = 0 / 0 = 0
Pr [PT=1 | CT=8] = 0.0 / 0.11 = 0.0
Pr [PT=1 | CT=9] = 0.0 / 0.03 = 0.0
Pr [PT=1 | CT=10] = 0.0 / 0.06 = 0.0
Pr [PT=1 | CT=11] = 0 / 0 = 0
Pr [PT=1 | CT=12] = 0 / 0 = 0

Pr [PT=2 | CT=1] = 0.0 / 0.06 = 0.0
Pr [PT=2 | CT=2] = 0.03 / 0.06 = 0.5
Pr [PT=2 | CT=3] = 0.0 / 0.11 = 0.0
Pr [PT=2 | CT=4] = 0.03 / 0.11 = 0.25
Pr [PT=2 | CT=5] = 0.0 / 0.06 = 0.0

Pr [PT=2 | CT=6] = 0.03 / 0.22 = 0.125
Pr [PT=2 | CT=7] = 0 / 0 = 0
Pr [PT=2 | CT=8] = 0.03 / 0.11 = 0.25
Pr [PT=2 | CT=9] = 0.0 / 0.03 = 0.0
Pr [PT=2 | CT=10] = 0.03 / 0.06 = 0.5
Pr [PT=2 | CT=11] = 0 / 0 = 0
Pr [PT=2 | CT=12] = 0 / 0 = 0

Pr [PT=3 | CT=1] = 0.0 / 0.06 = 0.0
Pr [PT=3 | CT=2] = 0.0 / 0.06 = 0.0
Pr [PT=3 | CT=3] = 0.06 / 0.11 = 0.5
Pr [PT=3 | CT=4] = 0.0 / 0.11 = 0.0
Pr [PT=3 | CT=5] = 0.0 / 0.06 = 0.0
Pr [PT=3 | CT=6] = 0.06 / 0.22 = 0.25
Pr [PT=3 | CT=7] = 0 / 0 = 0
Pr [PT=3 | CT=8] = 0.0 / 0.11 = 0.0
Pr [PT=3 | CT=9] = 0.03 / 0.03 = 1.0
Pr [PT=3 | CT=10] = 0.0 / 0.06 = 0.0
Pr [PT=3 | CT=11] = 0 / 0 = 0
Pr [PT=3 | CT=12] = 0 / 0 = 0

Pr [PT=4 | CT=1] = 0.0 / 0.06 = 0.0
Pr [PT=4 | CT=2] = 0.0 / 0.06 = 0.0
Pr [PT=4 | CT=3] = 0.0 / 0.11 = 0.0
Pr [PT=4 | CT=4] = 0.06 / 0.11 = 0.5
Pr [PT=4 | CT=5] = 0.0 / 0.06 = 0.0
Pr [PT=4 | CT=6] = 0.0 / 0.22 = 0.0
Pr [PT=4 | CT=7] = 0 / 0 = 0
Pr [PT=4 | CT=8] = 0.06 / 0.11 = 0.5
Pr [PT=4 | CT=9] = 0.0 / 0.03 = 0.0
Pr [PT=4 | CT=10] = 0.0 / 0.06 = 0.0
Pr [PT=4 | CT=11] = 0 / 0 = 0
Pr [PT=4 | CT=12] = 0 / 0 = 0
Pr [PT=5 | CT=1] = 0.03 / 0.06 = 0.5

Pr [PT=5 | CT=2] = 0.0 / 0.06 = 0.0
Pr [PT=5 | CT=3] = 0.03 / 0.11 = 0.25
Pr [PT=5 | CT=4] = 0.0 / 0.11 = 0.0
Pr [PT=5 | CT=5] = 0.03 / 0.06 = 0.5
Pr [PT=5 | CT=6] = 0.03 / 0.22 = 0.125
Pr [PT=5 | CT=7] = 0 / 0 = 0
Pr [PT=5 | CT=8] = 0.03 / 0.11 = 0.25
Pr [PT=5 | CT=9] = 0.0 / 0.03 = 0.0
Pr [PT=5 | CT=10] = 0.03 / 0.06 = 0.5

Pr [PT=5 | CT=11] = 0 / 0 = 0
Pr [PT=5 | CT=12] = 0 / 0 = 0

Pr [PT=6 | CT=1] = 0.0 / 0.06 = 0.0
Pr [PT=6 | CT=2] = 0.0 / 0.06 = 0.0
Pr [PT=6 | CT=3] = 0.0 / 0.11 = 0.0
Pr [PT=6 | CT=4] = 0.0 / 0.11 = 0.0
Pr [PT=6 | CT=5] = 0.0 / 0.06 = 0.0
Pr [PT=6 | CT=6] = 0.08 / 0.22 = 0.375
Pr [PT=6 | CT=7] = 0 / 0 = 0
Pr [PT=6 | CT=8] = 0.0 / 0.11 = 0.0
Pr [PT=6 | CT=9] = 0.0 / 0.03 = 0.0
Pr [PT=6 | CT=10] = 0.0 / 0.06 = 0.0
Pr [PT=6 | CT=11] = 0 / 0 = 0
Pr [PT=6 | CT=12] = 0 / 0 = 0

c> This algorithm provides perfectly secrecy. Proof:
Given, K ∈ [1…12], M or PT ∈ [1…6], and CT = (M * K) mod 13. CT ∈ [1…12].
Enc(k, m): c = m * k (mod 13)
Dec(k, c): m = c / k (mod 13)

Let's say we have arbitrary, m ∈ PT and c ∈ CT.

Pr [CT=c| PT=m] = Pr [K = c / m (mod 13)]
= Pr [K = k (mod 13)]
= 1/12

Pr [CT=c] = $\sum_{k∈K}$ Pr [K = k] · Pr [CT=c|K=k]
= $\sum_{k∈K}$ Pr [K = k] · Pr [PT = Dec(k, c)]
= $\sum_{k∈K}$ (1/12) * Pr [PT = c/k (mod 13)]
= (1/12) * $\sum_{m∈PT}$ * Pr [PT = m (mod 13)]
= 1/12

Therefore, we have Pr [CT = c| PT = m] = Pr [CT = c] for any m ∈ PT and c ∈ CT. Therefore, the explained algorithm provides perfect secrecy. It showed that ciphertext is independent of the plaintext. Also, we can conclude that for all m1, m2 ∈ PT and c ∈ C, we have Pr [CT = c| PT = m1] = Pr [CT = c| PT = m2] = 1/12.

Problem 7:

The simplest example to create M1 = 100*"A". So, all the letters in messages M1 map to only one plaintext that is A. Since, the key is of length 50, then we know that the cipher text will have a specific structure. The structure is that the C1 for C1[0:50] = C1[51:100] = key, or each ciphertext that appear in C1[0:50] will be repeated twice, and it will be the key.

Now, create M2 = 25* “ABCD”. So, the letters in the string “ABCD” are repeated 25 times in message M2 and they will map to some ciphertext. But, ciphertexts in C2[0:50] != C2[51:100] and the contents of C2[0:50] will not be repeated twice like it did for C1. But, most importantly for C2, if we have used the same key as M1 to encrypt M2, it is computationally infeasible to get the C2 being the same as C1, unless M1 == M2.

We know that M1 != M2, so I will prove it using contradiction. Let’s say k’ = C1-M1 mod 26 and k’ is used to encrypt M2. We know, C2 = M2 + k’ (mod 26)
=> M2 + (C1 – M1) mod 26 [substituting k’ = C1 – M1]

We want C2 = C1, so we can say, C2 = C1 = M2 + (C1 – M1) mod 26, or
=> C1 = M2 + (C1 – M1) mod 26
=> k’ – M1 = M2 + (C1 – M1) mod 26 [substituting C1 = k’ – M1]
=> M2 = k’ – C1 mod 26

We know k’ – C1 = M1 or M2 == M1. We know, that M2 != M1, that means k’ cannot be used to encrypt M2 and get C2 ==C1. Or in other words, if we have C2 == C1, we will have to create a new key, k’’ = c1 – m2 mod 26. Thus, in any case if C1 == C1, k’’ != k’, or if k’’ == k’, then C2 != C1.

From the above proof, we know that given two distinct plaintext M1 and M2, p[C2| M2] = 0 and p[C1| M1] > 0, if same key k is used for both C1 and C2. Thus, p[C1| M1] != p[C2| M2]. Thus, it violates the perfect-secrecy condition that, for all plaintext, M1 and M2, we should have p[CT=C|PT=M1] = p[CT=C|PT=M2].

Look at problem7.py to get the code that was used to generate the output.

```
M1:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
key:
LKCHEWJNAGRHAJXKBYXSNQTYSGVIQEQBYLTWOFBYVMZBSYUOVXLKCHEWJNAGRHAJX
KBYXSNQTYSGVIQEQBYLTWOFBYVMZBSYUOVX
ct:
JCHTKLFKWSWKVRDZMBJHUNGYVKMOXIBVCLNJRGPKSYVOGZFMAIJCHTKLFKWSWKVRD
ZMBJHUNGYVKMOXIBVCLNJRGPKSYVOGZFMAI
ct[0:50] =    JCHTKLFKWSWKVRDZMBJHUNGYVKMOXIBVCLNJRGPKSYVOGZFMAI
ct[51:100] =  JCHTKLFKWSWKVRDZMBJHUNGYVKMOXIBVCLNJRGPKSYVOGZFMAI
pt:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
M2:
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDA
BCDABCDABCDABCDABCDABCDABCDABCDABCD
key:
HFUMWWZMIKMEVWMLXZDOWHLFCJNDMHBXPFNKRUVCANUJPPHAAZVOIPNCKVNYMNAPY
JYSXHENSLUMCZFPOJXSZHNZOMMFJEGZKTKZ
ct:
MPXQNAXOCDIEJWRIUHNFYIEYMRZNPPBNBGVOQQLZVSYEUKBQYFTDBANDTDBXLADJK
WRUGWSYGDZTLXUKAGWJQBJMFNOQTXVGKHWU
ct[0:50] =    MPXQNAXOCDIEJWRIUHNFYIEYMRZNPPBNBGVOQQLZVSYEUKBQYF
ct[51:100] =  TDBANDTDBXLADJKWRUGWSYGDZTLXUKAGWJQBJMFNOQTXVGKHWU
pt:
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDA
BCDABCDABCDABCDABCDABCDABCDABCDABCD
```

Problem 8:

To prove, the statement “any cipher that offers perfect secrecy, the number of the possible keys must be at least as large as the number of plaintexts”, I am going to use pigeonhole principle and transformation concepts. If we have number of possible keys less than the number of plaintexts, that means for a given key and ciphertext, we have more than one possible preimage, or in other words there is no key for some (plaintext, ciphertext) pair. That means we will have to reuse keys, and thus be vulnerable to Kasisky Test.
Mathematically, we can rewrite this statement that because the number of possible keys is less than number of plaintext, there is a plaintext, p’, such that there is no key that maps the plaintext to ciphertext, as p[e(k, p’)=c]=0, which violates the perfect-secrecy condition that, for all plaintext, p1 and p2, we should have p[e(k, p1)=c] = p[e(k, p2)=c].

Another way of looking at it is that by trying all keys, we will at least have a duplicate plaintext, so we note that some plaintext will be more likely to occur than some other plaintext and the attacker is able to gain information about the parts of the key needed to decrypt the rest of the message.

Problem 9:

The RC4 algorithm is implemented in ‘problem9.py’. The library function that was used to generate the random IV, is os.urandom(). os.urandom has OS specific randomness source, such as `/dev/urandom`. `/dev/urandom` allow access to environmental noise collected from device drivers and other sources. Since, the environmental noise is very unpredictable, the IV generated is also very random.

Output:

Trial1:
```
pt:  Finally Done with CS6324 Homework
ct:
557A7D727F7F6A33577C7D7633647A677B33504025202127335B7C7E76647C6178
pt:  Finally Done with CS6324 Homework
```

trial2:
```
pt:  Finally Done with CS6324 Homework
ct:
2807000F0202174E2A01000B4E19071A064E2D3D585D5C5A4E2601030B19011C05
pt:  Finally Done with CS6324 Homework
```