Exercise 3

Chosen-plaintext attack(CPA)

Enigma was broken using both Chosen-plaintext attacks and known-plaintext attacks. We will describe the usage of Chosen-plaintext attacks. The british used gardening, arranging mine laying missions and letting the germans know on purpose. That way, they knew what the germans would send messages about, and therefore "choosing"the plaintext themselves. (Read about on wikipedia and $http://www.princeton.edu/achaney/tmve/wiki100k/docs/Chosen-plaintext_attack.html$)

Known-plaintext attack(KPA)

The A5/1 algorithm was broken using known-plaintext attacks. The breakers only had to know a few seconds of conversation to break the key. Found and read about at $http://en.wikipedia.org/wiki/A5/1$

Exercise 5

```
import time

def encrypt(plaintext, key):
    cipher = ['0'] * len(plaintext)
    for x in xrange(0, len(plaintext)):
        cipher[x] = str(ord(plaintext[x]) ^ ord(key[x]))
    return cipher

def bruteforce(plaintext, cip):
    cntr = 1
    brute_key = ['A'] * len(key)
    lostKey=[65]*16
    while True:
        for x in range(0,len(plaintext)):
            brute_key[x]=chr(lostKey[x])
        newCiph = encrypt(plaintext, brute_key)
        if newCiph == cip:
            print "Plaintext is: "+ ''.join(plaintext)+" the bruteforce text is: "+ '
            print "I ran: %d Times"%cntr
            break
        index = 15;
        lostKey[index]=lostKey[index]+1
        while index >= 0 and lostKey[index] >= 123:
            lostKey[index] = 65
            index=index-1
            if index < 0:
              break
            lostKey[index]=lostKey[index]+1
        cntr = cntr +1
```

```
if __name__ == '__main__':
    start = time.clock()
    key = ['A','A','A','A','A','A','A','A','A','A','A','A','A','z','A','A']
    plaintext = ['h','e','j','h','e','j','h','e','j','h','e','j','h','e','j','h']
    cip = encrypt(plaintext, key)
    bruteforce(plaintext, cip)
    stop = time.clock()
    print "Took", (stop - start), "sec"
```

We had a faster implementation in Java, but rewrote it in Python to make it shorter and trying to reach the 1 page limit. I will use our data from Java in the calculations. If you would like to see the Java program instead, let us know.
We only used characters from A-z in our key/plaintext
The program checked around 780000 keys each second. That gives us 2808000000 keys checked each hour. That gives us around $2^{44.5}$ keys checked in a year.
If we have a bot net of 1 million computers, we would be able to check $780000 \cdot 1 * 10^6 = 2^{39}$ keys each second, 2^{51} each hour and 2^{64} each year. Given our key is 16 long, with 52 possible characters at each spot, there is around 2^{91} possible combinations. That makes it possible to break keys up to 88 bits. Given that we could'nt break our 128 bit key with a million computers in a year, and that is if we have both plaintext and cipher.

Exercise 6

The unicity distance is calculated based on a simple substitution cipher, using english letters only. We see captial and lowercase as the same. The unicity distance is calculated by using following equation:

$$|UD| = \frac{log_2(|K|)}{R_L \cdot log_2(|P|)} \tag{1}$$

We use the following:
$|K| = 26!$
$|P| = 26$
$R_L = 0.7$

$$UD = \frac{log_2(26!)}{0.7 \cdot log_2(26)} = \frac{88.39}{0.7 \cdot 4.7} = 26.8 \tag{2}$$

So the average number of ciphertext characters required to eliminate alle spurious keys is around 27.

Exercise 9

The χ^2 test is performed on the given data. The four pairs of bit should occour an equal ammount of times, if the random number generator is perfect. But as it's a quite small sample size, there will be fluctations. The following data is given:

$$k_1(''11'') = 228 \quad e_1(''11'') = 250 \quad k_2(''10'') = 270 \quad e_2(''10'') = 250$$
$$k_3(''01'') = 271 \quad e_3(''01'') = 250 \quad k_4(''00'') = 231 \quad e_4(''00'') = 250$$

The following equation is used:

$$\chi^2 = \frac{(o_1 - e_1)^2}{e_1} + \frac{(o_2 - e_2)^2}{e_2} + \ldots \frac{(o_k - e_k)^2}{e_k} \tag{3}$$

$$\chi^2 = \frac{(228 - 250)^2}{250} + \frac{(270 - 250)^2}{250} + \frac{(271 - 250)^2}{250} + \frac{(231 - 250)^2}{250} = 6.744$$

Since the degree of freedom is 3 in our example, the probability value is:

$$P = \frac{1}{\Gamma\left(\frac{3}{2}\right)} \cdot \Gamma\left(\frac{3}{2}, \frac{6.744}{2}\right) = 0.08 = 8\% \tag{4}$$

A P-value of 0.05 or less is regarded as statistically significant. Therefore our example is regarded as non significant.

Exercise 11

I've made a encryption tool:

```
int initVector;
int key[] = { 13, 4, 3, 12, 1, 0, 8, 10, 14, 6, 9, 15, 11, 2, 5, 7 };
int plaintext[] = { 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 };
int cipher[16];
int bitSize = 16;

void encrypt(int *plainText, int initVector){

    cipher[0] = key[initVector ^ plainText[0]];

    for (int i = 1; i < bitSize; i++) {
        cipher[i] = key[cipher[i - 1] ^ plainText[i]];
    }
}

int main(int argc, const char * argv[])
{

    for (int i = 0; i < bitSize; i++) {
        initVector = i;

        encrypt(plaintext, initVector);
        printf("InitVector: %d Cipher: ", initVector);
        for (int j = 0; j < bitSize; j++) {
            printf("%d ", cipher[j]);
        }
        printf("\n");
    }

}
```

The tool encrypts the plaintext, using every 4-bit initialisation vector. The following console output is given:

```
InitVector: 0 Cipher: 12 7 1 3 13 5 8 15 11 14 2 4 10 6 0 12
InitVector: 1 Cipher: 3 13 5 8 15 11 14 2 4 10 6 0 12 7 1 3
InitVector: 2 Cipher: 4 10 6 0 12 7 1 3 13 5 8 15 11 14 2 4
InitVector: 3 Cipher: 13 5 8 15 11 14 2 4 10 6 0 12 7 1 3 13
InitVector: 4 Cipher: 10 6 0 12 7 1 3 13 5 8 15  11 14 2 4 10
InitVector: 5 Cipher: 8 15 11 14 2 4 10 6 0 12 7 1 3 13 5 8
InitVector: 6 Cipher: 0 12 7 1 3 13 5 8 15 11 14 2 4 10 6 0
InitVector: 7 Cipher: 1 3 13 5 8 15 11 14 2 4 10 6 0 12 7 1
InitVector: 8 Cipher: 15 11 14 2 4 10 6 0 12 7 1 3 13 5 8 15
InitVector: 9 Cipher: 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
InitVector: 10 Cipher: 6 0 12 7 1 3 13 5 8 15 11 14 2 4 10 6
InitVector: 11 Cipher: 14 2 4 10 6 0 12 7 1 3 13 5 8 15 11 14
InitVector: 12 Cipher: 7 1 3 13 5 8 15 11 14 2 4 10 6 0 12 7
InitVector: 13 Cipher: 5 8 15 11 14 2 4 10 6 0 12 7 1 3 13 5
InitVector: 14 Cipher: 2 4 10 6 0 12 7 1 3 13 5 8 15 11 14 2
InitVector: 15 Cipher: 11 14 2 4 10 6 0 12 7 1 3 13 5 8 15 11
```

We see that every unique initialisation vector, produces a different output of the cipher. We also see that, when the initialisation vector is 9 every number in the cipher text is also 9. This is expected, as 3 XOR 9 = 10 and a p = 10 produces a c = 9 creating a loop.