6.857 Homework	**Problem Set 1**	# 1-2. One-time pad
Qian Long		
Joseph W Presbrey		
Mikhail Rudoy		February 20, 2014

1-2a)

Given two ciphertext C_1 and C_2 encoding using the same one-time-pad P, we can deduce the original messages M_1 and M_2 by xoring the ciphertexts, as follows.

$$\begin{aligned} M_1 \oplus P &= C_1 \\ M_2 \oplus P &= C_2 \\ M_1 \oplus M_2 &= C_1 \oplus C_2 \end{aligned}$$

To find the secret words we can first compute $C_1 \oplus C_2$ using the given values. Then, using the above relation, we simply have to find a pair of 8-character words M_1 and M_2 such that $M_1 \oplus M_2$ is the computed value.

Given the assumption that the secret words are common 8-character English words, we can take a list of common 8-character words from the internet and write a python script to look for the pair of words that, when xor-ed together, equals $C_1 \oplus C_2$. We optimize the performance of the search by first computing $M_1 \oplus C_1 \oplus C_2 = M_2$ for each M_1 in the list of words and storing the results in a hash table. Then we scan the word list again and simply check if the each word is in the hash table, taking $O(1)$ time for each check. If we find a word in the hashtable, then the pair M_1 and M_2 are found. Thus, this script runs in linear time.

The two words used for the specific code given are “security” and “networks”.

Python code:

```
def ascii_word(string):
  output = []
  for char in string:
    output.append(ord(char))
  return output


def xor_word(array1, array2):
  output = []
  for i in xrange(len(array1)):
    output.append(array1[i] ^ array2[i])
  return output

def main():
  c1 = [0xe9, 0x3a, 0xe9, 0xc5, 0xfc, 0x73, 0x55, 0xd5]
  c2 = [0xf4, 0x3a, 0xfe, 0xc7, 0xe1, 0x68, 0x4a, 0xdf]
  result = xor_word(c1, c2)
  # str rep of ascii bytes
  resultstr = ''.join([str(x) for x in result])

  f = open('words.txt', 'r')
  words = []
  for line in f.readlines():
    words.extend(line.strip().split(' '))

  # find 2 words such that w1 ^ w2 == result
  # put result ^ w into hashtable => result ^ w1 = w2
  hashtable = {}
  for i in xrange(len(words)):
    m2 = ''.join([str(x) for x in xor_word(result, ascii_word(words[i]))])
    hashtable[m2] = words[i]

  for i in xrange(len(words)):
    ascii_str = ''.join([str(x) for x in ascii_word(words[i])])
    if ascii_str in hashtable:
      print "found match:"
      print hashtable[ascii_str]
      print words[i]
      break
  return

if __name__ == "__main__":
  main()
```

1-2b)

For this part, we were given 10 ciphertexts, $C_1, C_2, ..., C_{10}$ which are the encodings of messages $M_1, M_2, ..., M_{10}$ with the same key. Our solution relies on the following idea:

Suppose we can guess part of the message. As an example, suppose we can correctly guess the value of m_{ji}. We know that

$$c_{ji} = m_{ji} \oplus ((p_i + c_{j(i-1)}) \mod 256)$$

We have guessed m_{ji} and we have access to the ciphertexts, so we can compute

$$\begin{aligned}
(m_{ji} \oplus c_{ji}) - c_{j(i-1)} &= (m_{ji} \oplus (m_{ji} \oplus ((p_i + c_{j(i-1)}) \mod 256))) - c_{j(i-1)} \\
&= ((m_{ji} \oplus m_{ji}) \oplus ((p_i + c_{j(i-1)}) \mod 256)) - c_{j(i-1)} \\
&= ((p_i + c_{j(i-1)}) \mod 256) - c_{j(i-1)} \\
&= p_i \mod 256
\end{aligned}$$

Having computed p_i, we can then compute m_{ki} for every k.

When we guess a character, the result is that we can immediately compute what the characters in the same position of the other messages are. These values we compute will be between 0 and 255. However, the characters used are ASCII-encoded, and are therefore between 0 and 127. This is a good "sanity check" for a guess. If the guess is correct, all of the corresponding characters in the other messages are necesserily correct, and the guess passes the sanity check. If the guess is wrong, it might pass, and it might not.

There is a convenient (and completely theoretically unsupported) calculation that can be done to get a feeling for how good this test is: If the guess is wrong, we will (with no theoretical basis) assume that the 9 corresponding characters from the other messages are chosen uniformly at random from the range of possible values. Then there is a $\frac{1}{512}$ chance that this test accepts the guess as potentially valid.

In practice, the test is not so successful, but it does reject a significant fraction of wrong guesses.

Instead of guessing a single character, what we actually do is guess a sequence of characters. For example, it is quite reasonable to assume that English text will contain the strings "to" and "the" (or better yet, the strings " to " and " the "). Once we decide on a string which we believe is a substring of one of the messages, we can attempt to match that string with every possible substring of the messages of the correct length. Each possible positioning of the string is one guess. A large majority of the guesses are generally rejected by the above test, and the remaining collection of guesses is small enough that it can be easily processed by a person.

Of the guesses remaining, it tends to be easy to see which are correct and which are incorrect. The correct ones look like English text, and the others look like gibberish. The property of "English-ness" is certainly one that we can train a computer to look for (i.e. we generally don't expect any character except "u" after a "q"), but we chose to not invest the effort into programming a complete set of rules, and instead chose to use our own pattern detection abilities.

Once we have accepted a certain guess as correct, the pieces of the other messages that are revealed allow us to easily decide what words to try to match next (i.e. seeing "crypt" could suggest trying

to match "cryptography" with the messages). In this way, the regions that we have guessed can be expanded until the entire messages are revealed.

The code we wrote does exactly this. It allows a user to keep an archive of partial answers. Each partial answer is a list of characters believed to be the characters of the messages (or blanks). In addition to functions for exchanging partial answers between the archive and the "current answer" and a function which prints the contents of an archived partial answer, the code allows the user to enter a string which the program then tries to match with the messages in every possible location. Every guess for the location of that string in the message is either rejected because it fails the test described above, rejected because it does not coincide with the current answer, or given to the user to accept or reject.

The program, with suitable input, successfully decoded the given ciphertexts.

Here are the messages, in order:

```
We stand today on the brink of a revolution in cryptography.
Probabilistic encryption is the use of randomness in an encr
Secure Sockets Layer (SSL), are cryptographic protocols that
This document will detail a vulnerability in the ssh cryptog
MIT developed Kerberos to protect network services provided
NIST announced a competition to develop a new cryptographic
Diffie-Hellman establishes a shared secret that can be used
Public-key cryptography refers to a cryptographic system req
The keys used to sign the certificates had been stolen from
We hope this inspires others to work in this fascinating fie
```

The following is a list of numbers, each representing a byte of the pad used:
119, 75, 116, 51, 85, 113, 72, 105, 76, 78, 114, 79, 84, 49, 71, 101, 71, 88, 116, 78, 113, 102, 113, 87, 84, 65, 51, 55, 99, 56, 107, 69, 116, 105, 110, 109, 97, 113, 79, 106, 122, 68, 66, 98, 77, 72, 112, 72, 55, 53, 104, 54, 99, 71, 87, 97, 68, 98, 112, 49

The code of the python program used is reproduced below:

```
ctxts = [[32, 14, 162, 166, 143, 97, 199, 84, 128, 186,
67, 246, 43, 37, 76, 222, 75, 131, 131, 185,
79, 149, 100, 201, 116, 219, 101, 188, 112, 206,
25, 63, 147, 142, 153, 112, 190, 67, 231, 37,
246, 85, 249, 123, 161, 135, 215, 124, 193, 143,
135, 201, 67, 237, 54, 246, 74, 196, 77, 80],
[39, 0, 27, 44, 224, 51, 18, 23, 10, 43,
233, 81, 198, 215, 123, 142, 182, 124, 137, 167,
108, 187, 67, 244, 104, 192, 128, 151, 142, 174,
124, 225, 32, 250, 13, 90, 212, 35, 82, 206,
41, 3, 33, 236, 84, 242, 7, 60, 0, 21,
20, 36, 167, 143, 136, 201, 104, 164, 119, 218],
[36, 10, 29, 37, 8, 28, 68, 254, 37, 16,
233, 93, 197, 133, 236, 29, 5, 36, 253, 57,
138, 216, 26, 34, 58, 82, 169, 192, 66, 8,
22, 123, 140, 135, 140, 137, 158, 96, 200, 64,
219, 111, 217, 82, 252, 100, 164, 158, 186, 155,
108, 193, 75, 254, 38, 167, 159, 105, 184, 157],
```

```
[35, 6, 19, 53, 170, 127, 168, 114, 203, 116,
131, 188, 100, 181, 139, 153, 140, 136, 220, 78,
218, 52, 196, 114, 170, 203, 159, 246, 47, 18,
17, 56, 201, 64, 207, 94, 214, 43, 19, 9,
250, 30, 9, 5, 114, 206, 86, 251, 18, 52,
239, 77, 144, 180, 121, 163, 151, 141, 146, 164],
[58, 204, 20, 103, 216, 44, 2, 14, 54, 235,
45, 25, 9, 26, 42, 234, 67, 249, 8, 36,
250, 19, 164, 143, 140, 237, 80, 245, 55, 27,
227, 75, 203, 20, 236, 60, 233, 45, 19, 15,
226, 6, 59, 248, 55, 9, 16, 59, 23, 63,
135, 205, 66, 230, 75, 197, 109, 170, 126, 143],
[57, 205, 18, 17, 70, 214, 112, 183, 108, 207,
47, 29, 20, 33, 72, 204, 51, 232, 51, 236,
45, 246, 19, 3, 35, 13, 47, 8, 75, 247,
13, 114, 130, 142, 138, 146, 159, 127, 190, 8,
227, 7, 39, 236, 78, 182, 69, 255, 79, 244,
40, 49, 243, 72, 254, 47, 27, 20, 231, 56],
[51, 23, 237, 70, 242, 6, 99, 132, 181, 111,
141, 177, 100, 251, 98, 162, 154, 134, 155, 139,
144, 159, 99, 210, 67, 247, 10, 32, 163, 168,
123, 161, 103, 181, 71, 148, 134, 146, 130, 158,
125, 181, 215, 77, 242, 91, 191, 39, 61, 19,
21, 107, 172, 150, 205, 91, 236, 43, 255, 16],
[39, 7, 25, 32, 28, 238, 27, 239, 94, 213,
103, 213, 91, 245, 76, 197, 99, 220, 34, 17,
242, 48, 216, 15, 17, 55, 12, 38, 251, 64,
139, 164, 119, 192, 79, 156, 158, 125, 181, 111,
157, 142, 183, 107, 217, 81, 169, 152, 172, 193,
90, 233, 63, 242, 44, 224, 4, 20, 225, 99],
[35, 6, 31, 114, 172, 120, 185, 81, 189, 126,
131, 183, 111, 128, 179, 119, 158, 133, 144, 185,
68, 138, 143, 142, 135, 232, 120, 202, 95, 227,
39, 10, 23, 227, 48, 233, 47, 211, 2, 4,
31, 7, 105, 169, 147, 190, 64, 168, 172, 149,
146, 164, 98, 199, 62, 249, 79, 222, 35, 116],
[32, 14, 162, 189, 125, 158, 131, 204, 108, 210,
45, 15, 67, 29, 10, 28, 19, 2, 4, 55,
219, 97, 189, 96, 220, 120, 217, 99, 230, 106,
186, 223, 36, 226, 34, 228, 101, 191, 96, 234,
16, 60, 23, 10, 119, 217, 40, 3, 89, 231,
33, 54, 237, 93, 218, 92, 128, 132, 157, 171]]

answer = [[None for i in range(len(ctxts[0]))] for i in range(len(ctxts))]
archived_answers = {}

def print_answer(arg = None):
    if arg:
        a = archived_answers[arg]
    else:
```

```
        a = answer
    for l in a:
        s = ""
        for c in l:
            if c == None:
                s+="?"
            else:
                s+=c
        print s

def archive(val):
    archived_answers[val] = [l[:] for l in answer]

def restore(val):
    global answer
    archive("undo")
    answer = [l[:] for l in archived_answers[val]]

def Help():
    print """use the following commands to crack the code:
Help() to get help
print_answer() to print what you currently think the messages look like
archive(arg) to save your current guess under the key arg
restore(arg) to restore a previous level of knowledge (i.e. if you mess up)
print_answer(arg) to print what a particular saved guess looks like
look_for_text(txt) to guess part of the message and see where it could fit
 - enter y or yes to accept a potential location of the guessed message text
 - enter anything else to reject

the archive comes preset with a zero-knowledge setup saved under "restart",
so you can use restore("restart") to restart"""

Help()
archive("restart")
```

```
def look_for_text(txt):
    global answer
    original = answer
    for i in range(len(ctxts[0]) - len(txt) + 1):
        for m in range(len(ctxts)):
            valid = True
            useful = False
            answer_copy = [l[:] for l in answer]
            for j in range(len(txt)):
                if answer_copy[m][i+j] == None:
                    useful = True
                if answer_copy[m][i+j] != None and answer_copy[m][i+j] != txt[j]:
                    valid = False
                answer_copy[m][i+j] = txt[j]
            if not valid or not useful:
                continue
            vals = [ctxts[m][i + j] ^ ord(txt[j]) for j in range(len(txt))]
            padpart = []
            for j in range(len(txt)):
                if i+j == 0:
                    padpart.append(vals[j])
                else:
                    padpart.append((vals[j] + 256 - ctxts[m][i+j-1])%256)
            s = ""
            valid = True
            for otherm in range(len(ctxts)):
                if otherm != m:
                    for j in range(len(txt)):
                        if i+j == 0:
                            c = padpart[j] ^ ctxts[otherm][i+j]
                        else:
                            c = (padpart[j] + ctxts[otherm][i+j-1])%256
                            c = c ^ ctxts[otherm][i+j]
                        valid = valid and (c < 128)
                        if answer_copy[otherm][i+j] and ord(answer_copy[otherm][i+j]) != c:
                                valid = False
                        answer_copy[otherm][i+j] = unichr(c)
            if valid:
                for l in answer_copy:
                    s = ""
                    for c in l:
                        if c == None:
                            s+="?"
                        else:
                            s+=c
                    print s
                if raw_input() in ["yes", "y"]:
                    answer = answer_copy
    if original != answer:
        archived_answers["undo"] = original
```