Homework: One-time Pad Encryption in Machine Code
Programming for Scientists
Submit via Autolab

Reading

Read the introduction and first two sections of the Wikipedia page on modular arithmetic: `https://en.wikipedia.org/wiki/Modular_arithmetic`

Learn about two's complement for using an n-digit hexadecimal number to represent both positive and negative integers at `http://www.tfinley.net/notes/cps104/twoscomp.html`.

Assignment

Set up

Download and run X-TOY from its website: `http://introcs.cs.princeton.edu/xtoy/`.

If you have not done so already, make sure today that you can run X-TOY on your machine.

One-time Pad Encryption

Often we have a message that we want to keep secret from some third party. For example, you want to email someone our social security number, but you don't want anyone who intercepts our email to be able to read it. There are many such encryption schemes, and designing good encryption schemes and secure transmission protocols is a very active area of research. In this assignment, we will consider one of the first schemes proposed for encryption, called **one-time pad**.

Suppose we have a *message* t, encoded as stream of bits:

```
t = 0101010000000111111100010101
```

we also have a *secret key* s of the same length:

```
s = 1111010100011111100101010101
```

we could encrypt t by computing `r = t XOR s`[1], where `XOR` is the function that sets the ith bit of r based on the ith bits of t and s according to the following table:

s_i	t_i	r_i
0	0	0
0	1	1
1	0	1
1	1	0

[1]Note: it is public information that we are using `XOR`.

In other words, r_i is 1 if and only if exactly one of t_i and s_i are equal to 1. (If 1 represents "true" and 0 represents "false", you can think of the above table as a "truth table" where `s XOR t` is true whenever one, but not both, of s and t is true. For this reason, `s XOR t` is equivalent to `(s || t) && (!s || !t)`. In the example above,

```
t = 01010100000001111100010101
s = 11110101000111100101010101
r = 10100001000110001100100000
```

The bits r can be used as our encrypted message. The great thing about XOR is that if we know s and r we can recover t. Below w is `r XOR s`:

```
r = 10100001000110001100100000
s = 11110101000111100101010101
w = 01010100000001111100010101
```

Here $w = t$!

So: if Alice and Bob each know s, Alice can send Bob an encrypted message by computing `r = t XOR s`. Bob can then read it by computing `t = r XOR s`.[2]

The next question is how to generate a good secret key s? One answer is that we can choose it randomly.

Random Number Generators

We've used the function `rand.Int()` for example to generate random numbers for several of our assignments.

As we have seen in class, random numbers generated by a computer are not truly random, they just "look" random, i.e., there is no obvious pattern to them. Programming languages generate a sequence of random numbers by repeatedly applying a function f:

$$R_0, R_1 = f(R_0), R_2 = f(R_1), \ldots, R_n = f(R_{n-1})$$

In class, we saw one example of f as the middle-square function, and we will see another possible definition of f in a moment. R_0 is the *seed*, and when you call `rand.Seed()`, you are setting the value of R_0. When you ask for a new random number (using, e.g., `rand.Int()`), Go takes the current random number R_i and applies the function f to it to get the next number R_{i+1}.

What is a good choice of f? This is a very deep question that researchers have worked on for decades and that is still relevant today (don't believe me? Check out `https://en.wikipedia.org/wiki/NIST_SP_800-90A`). Even deciding what we mean by "a random sequence of numbers" is a point of contention, as is the philosophical question about whether randomness exists in the universe (see `https://en.wikipedia.org/wiki/Uncertainty_principle`). One thing we know that we want from f is that it produces a complex sequence of numbers R_i using a simple rule f. (Again, a theme of the class pops up: complex behavior from simple rules.) Several classes of functions seem to exhibit this property, and we'll discuss one: linear congruent generators.

[2] How do we know no one else can read it? Because we can create *any* bit string t' from r using the appropriate s': if I want t'_i to be 1 I choose s'_i to be 0 or 1 according to the value of r'_i and the table above; if I want t'_i to be 0, I can similarly choose s'_i to make that happen. So if any decoded message can be obtained by choosing the right s', then even trying all $2^{|s|}$ possible s wouldn't tell me which of the $2^{|s|}$ decoded messages was the right one.

Let a, m, c be integers. We define f as:

$$f(R) = (aR + c) \mod m$$

Here mod is the remainder operator, expressed in Go as `%`. Not all choices of a, m, and c give a good function f, but it turns out that the choice

$$a = 2^4 + 1 \tag{1}$$
$$m = 2^{16} \tag{2}$$
$$c = 1 \tag{3}$$

gives a reasonably good (though not the best) sequence of random-looking numbers. (Note: these constants were chosen for ease of implementation rather than for getting a good random number generator — if you were to write an industrial strength random number generator, you should use different constants.)

What you should do

You will write a program in X-TOY machine language to (1) read in a message from standard input, (2) generate a random secret key, and (3) output both the encrypted message and the key. Your program will also be able to *decrypt* these messages.

Input

Your program will receive input on standard input. The first number you read will be a nonzero seed for the random number generator if you are supposed to encrypt and 0 if you are supposed to decrypt. The next number will be the message length n in words. The next n words will be the message in 16-bit chunks (either t or r depending on if you are encrypting or decrypting). Finally, if you are *decrypting*, the next n words will be the secret key s. In other words, standard input will be a stream of words of the following format:

```
0 or R0     ; 0=decrypt; non-zero = encrypt with seed R0
n           ; the number of words in the message
t_1         ; the first word of the message
...
t_n         ; the last word of the message
s_1         ; the first word of the key if decrypting
...
s_n         ; the last word of the key if decrypting
```

If you are encrypting, you should use the linear congruent random number generator described above to generate a random secret key s of length n words. You should then `XOR` s with the input message to get r. To standard output, you should then write: the number 0, then n, then the n encrypted words of the message, and then the n words of the key. Notice that this output format is exactly what is expected for the decryption stage.

If you are decrypting, you should `XOR` the s and r that you read in from the input to generate t. You should then write to standard output: the number n, and then the n words of t. Notice that this output format is exactly what is expected for the encryption stage, except for the seed.

You can assume that $n < 32.$

Tips on how to start

First, play around with X-TOY to get a feel for how you write programs. Do this early on.

Then, write the random number generator "function" that generates random numbers. Compare your X-TOY output with the same function written in Go.

Then, write the code to read the rest of the input for encryption, and perform the encryption using the random bits from your random number generator. Again, if needed, compare with the same function written in Go. You should probably write this using a function `XOR(n, location1, location2, location3)` that `XOR`s the n words at `location1` and `location2` and stores them in location `location3`.

Next, write the decryption code: you can again use your `XOR(n, location1, location2, location3)` function. Make sure that if you encrypt, then decrypt, that you obtain the original message.

Important Notes

First, X-TOY may sometimes interpret hexadecimal numbers as negative. Please ignore this; the same binary string can be interpreted as either an unsigned or signed integer according to the reading. Second, when running your program, please use sim mode instead of debug mode; this will prevent your getting an annoying error of the form "A runtime error has occurred at address `XX`: The result of an arithmetic [sic] operation was not between -32768 and 32767."

Learning outcomes

After completing this homework, you should have

- learned about how a CPU works and what is actually going on inside the computer
- gained an appreciation for what the Go compiler is doing
- learned about random number generators
- learned about one-time-pad encryption