0368.3049.01 **Fall 2009**

Introduction to Modern Cryptography

Benny Chor and Rani Hod

Assignment #1, version 2.1

Published Tues., October 20, 2010, and in revised form on Oct. 26. Due Tues., November 10, in Rani Hod's mailbox (second floor, Schreiber building).
This assignment contains three "dry" problems and three "wet" ones. Efficient solutions are always sought, but a solution that works inefficiently is better than none. The answers to the the "wet" problems should be given as the output of a `Sage`, `Maple`, or `WolframAlpha` session.
On Oct. 28 we will have some demos of running `Sage`.

Problem 1. Let p be a 128-bit prime and let Z_p be the set of integers $\{0, \ldots, p-1\}$. Consider the following encryption scheme. The secret key is a pair of integers $a, b \in Z_p$ where $a \neq 0$. An encryption of a message $M \in Z_p$ is defined as:

$$E_{a,b}(M) = aM + b \bmod p$$

(a) Show that when E is used to encrypt a *single* message $M \in Z_p$, the system is a perfect cipher. (For a definition, refer to notes from the first lecture.)

(b) Show that when E is used to encrypt *two* messages $M_1, M_2 \in Z_p$, the system is *not* a perfect cipher.
Hint: Consider the case $M_1 = M_2$.

(c) Show that a known plaintext attack with just two pairs of plaintext/ciphertext $C_i = E_{a,b}(M_i)$ $(i = 1, 2)$ can recover the secret key a, b with high probability.

Problem 2. The following is a special case of a permutation cipher: Let m, n be positive integers. Partition the plaintext to segments of nm letters each. Write down each plaintext segment by *rows* in an n-by-m matrix. The ciphertext is created by going over the *columns* of the matrix. For example, if $n = 3,\ m = 4$ the plaintext "cryptography" will lead to the matrix

$$\begin{array}{cccc} \text{c} & \text{r} & \text{y} & \text{p} \\ \text{t} & \text{o} & \text{g} & \text{r} \\ \text{a} & \text{p} & \text{h} & \text{y} \end{array}$$

and the ciphertext will be "ctaropyghpry".

(a) Decipher the ciphertext (generated in the abovementioned way, not necessarily with the same m and n) "myamraruyiqtenctorahroywdsoyeouarrgdernogw".

(b) Describe an effective method for deciphering long enough ciphertexts, encrypted by applying a regular substitution cipher first, followed by a permutation cipher as above. Limit your answer to no more than 8 lines.

Problem 3. The file `cipher1.txt` contains a message encrypted by a simple substitution cipher. The original message language is Hebrew and only characters in the Hebrew alphabet are encoded, leaving punctuation and whitespace intact.

(a) First we collect statistics about the Hebrew language. Get yourself a nice long Hebrew document (see, for example, http://benyehuda.org/) and estimate the order of the letter frequencies.

(b) Do the same for the enciphered text and try to match the statistics to those you have collected. It is expected to approximately match the original message, so can you now read it? Try to explain what you got.

(c) The message is known to contain somewhere in it the text "YALDI HATZACH". Does this help?

Note: the cipher, as well as texts in project Ben-Yehuda, can be read using the following `Sage` command.

```
m = open('filename.txt').read().decode('cp1255')
```

Problem 4. RTAU (an Internet music station) wishes to broadcast streamed music to its subscribers. Non-subscribers should not be able to listen in. When a person subscribes she is given a software player (which cannot be tempered with) with a number of secret keys embedded in it. RTAU encrypts the broadcast using a symmetric cryptosystem (private key) with a 128-bit key, K. The secret keys in each legitimate player can be used to derive K and enable legitimate subscribers to tune in. When a subscriber cancels her subscription, RTAU will encrypt future broadcasts using a different key K'. All legal subscribers should be able to derive K' , while the canceled subscriber should not.

(a) Suppose the total number of potential subscribers is less than $n = 10^5$. Let $R_1, R_2, \ldots, R_n$ be n random independent values, 128 bits each . The player shipped to subscriber number u contains all the R_i's except for R_u (*i.e.* each player contains 99999 keys). Let S be the set of currently subscribed users. Show that RTAU can construct a key K, used to encrypt the broadcast, so that every subscriber in S can derive K (from the R_i's in her player), while any single subscriber outside of S cannot derive K. You may assume that the set S is known to everyone (e.g. it is a plain part of the broadcast) . Briefly explain why your construction satisfies the required properties.

(b) Is your construction in part (a) collusion resistant? That is, can two canceled subscribers combine the secrets embedded in their player to build a new operational player?

Remark: Much better solutions to this problem exist.

Problem 5. In this problem we will investigate the run time behaviour of Euclid's greatest common devisor (gcd) algorithm, and while doing so, start becoming aquainted with `Sage`. As usual, if you are already familiar with `Maple` or `WolframAlpha`, you are welcome to write your code using these languages, but the hints below refer to Sage.

Repeat the following one thousand (1000) times for every integer n in the range $5 \leq n \leq 30$. Choose at random a pair of integers a, b with $a \geq b$ in the range $2^n < b \leq a \leq 2^{n+1}$. Compute $\gcd(a, b)$ using just `Sage`'s mod operator (for example, $\text{mod}(32, 17) = 15$). *Count* the number of mod operations, and record it. Compute the *average* and the *maximum* of this count for the different values of n, and separately plot the two as a function of n.

Submit your code, the plots, the pair a, b attaining the maimum count per n, and your estimates of both functions (e.g. $2^{n/2}, \sin(n), 3.7n^2$, etc.).

Some `Sage` hints:

- `randint(a,b)` returns a random integer x in the range $a \le x \le b$. Alternatively, `getrandbits(n)` generates a long int with n random bits (but note this int will have its most significant bit equal 0 half the time).
- The following small piece of code demonstrates one form of loop structure. It generates and prints 10 pairs of random integers in the range $[33, 64]$.

```
for k in range(10):
    print (randint(33,64), randint(33,64))
```

For plotting, generate a list of pairs of the form $(n, f(n))$ using `Sage` list operations and plot it using `point` or `plot_step_function`:

```
P = [(x, x**2) for x in range(10)]
point(P); plot_step_function(P)
```

Problem 6. In this problem we will become familiar with finite fields $GF(p^k)$ where $k > 1$. Specifically, we will look at the field $GF(2^4)$.
Find an irreducible polynomial $f(x)$ of degree 4 over the base field of characteristic 2, $\mathbb{Z}_2$. Implement the field $GF(2^4)$ in `maple` using the two statements

```
G16 := GF(2,4,f(x)); u := G16[ConvertIn](x);
```

Or in `Sage` using the commands

```
x = polygen(GF(2)) # make x the variable of the polynomial field
f = x^100 + x + 1  # insert your favorite polynomial here
assert f.is_irreducible()
G16.<u> = GF(2**4, modulus=f) # u is the generator's name
```

Once this is done, write a small loop which prints out all the primitive elements (multiplicative generators) in $GF(2^4)$. How many are there? The situation here is quite different than that of $GF(2^5)$. Briefly explain why. (in `maple` use `ConvertOut` to get a canonical representation of field elements, with higher degree momonials to the left)

Pick *at random* a 5 digit number a and a 6 digit number b that are relatively prime. Using just `Sage`/`Maple`'s mod command, run the Euclid gcd algorithm on your a and b. How many mod steps did it take? Now run the extended gcd algorithm (again, employing just mod operations) and compute the multiplicative inverse of a in Z_b.
Remark: both `Sage` and `Maple` have a built-in command for extended gcd. `Sage` has `xgcd`, taking a pair (x, y) and returning a triple (g, a, b) such that $g = ax + by = \gcd(x, y)$. `Maple` has `igcd`, used as `igcd`(x,y,'a','b'), which returns $g = \gcd(x, y)$ and assigns a and b to the values satisfying $g = ax + by$. You can use this to *verify* your `mod` computations.