Institut for Matematik og Datalogi
Syddansk Universitet

November 19, 2010
JFB

Introduction to Computer Science
E10 – Lecture 13

Lecture, November 8, 8:15–10, U37

We continued with chapter 12 in the textbook, giving an example using Bare Bones. We then covered security from sections 3.5 and 4.5 and introduced cryptography. We discussed both symmetric key and public key cryptography, particularly how they are used together. Information about this is available from the two links on the course's homepage about PGP (the 5th and 6th lines from the bottom).

Lecture, November 11, 14:15–16, U71

We will finish chapter 12.

Lecture, November 15, 8:15–10, U37

We will cover sections 6.6 and 6.7 of chapter 6.

Discussion section: November 19, 8:15–10, Terminal Room

Discuss the following problems in groups of two or three.

The best known public key cryptographic system, RSA, was presented in lectures. It is one of the systems included in PGP and GPG. Its security is based on the assumption that factoring large integers is hard. (The system you are using in GPG is based on discrete logarithms, rather than factoring, but the problems are similar in many ways. The factoring is easier to understand and test in Maple.)

A user's public key consists of a large integer n (currently numbers with at least 1024 bits are recommended, and 2048 is being recommended by many experts) and an exponent e. The integer n should be a product of two prime numbers p and q, both of which should be about half as long as n. Thus, in order to implement the system it must be possible to find two large primes and multiply them together in a reasonable amount of time. For the security of the system, it must be the case that no one who does not know p or q could factor n.

At first glance this seems strange, that one should be able to determine if a number is prime or not, but not be able to factor it. However, there are algorithms for testing primality, which can discover that a number is composite (not prime) without finding any of its factors. (The ones most commonly used are probabilistic, so they could with small probability declare a composite number prime; the probability of this happening can be made arbitrarily small.)

Using Maple, you should try producing primes and composites and try factoring.

In the following, you will be using the program `xmaple`:

1. Small numbers.

   Start your Maple program, using the command `xmaple`. Type `restart;` at the beginning to make it easier to execute your worksheet after you have made changes. (You can execute the worksheet after changes from `Execute` in the `Edit` menu.)

   Use *Help* to find out about the function *ithprime*. Experiment to find out approximately how big a prime it can find. When it cannot find such a big prime, you can use the STOP button in order to continue (it is a hand in a red background). To assign a value to a variable, you use the assignment operator `:=`; for example `x:= ithprime(4);`. Multiply two of the large primes it finds together, and try to factor the result, using the function *ifactor*. Notice how quickly the factors are found for these small numbers. (Large numbers are clearly necessary for security.)

2. Finding larger primes.

   In order to find good prime factors p and q for use in RSA, one can choose random numbers of the required length and check each one for primality until finding a prime.

Maple contains a function *isprime* which will test for primality. Try it on some some small numbers, such as 3, 4, 7, 10. Maple has another function *rand* which returns a random 12-digit number. Try typing `x:=rand();` and check if your result is prime. Rather than executing these commands until you find a prime, you can use a *while loop*. You want to continue creating new random numbers until you get a prime, so you can type `while (not isprime(x)) do`, followed by `x:=rand();` and `end do;`. To get this to function together, you can either use the `ESC` key to input a `while` loop, or you can use `CTRL ENTER` instead of just `ENTER` to start a new line. How many different values were chosen before a prime was found? (I got 33, but you could get another number.) Now create a second prime called y (remember that y will need some value before your start your *while loop*). Multiply x and y together and try factoring the result. This should also go relatively quickly.

3. Finding even larger primes.

   To get random numbers which are twice as long, you can create three random numbers a, b and c and create $10^{24} * a + 10^{12} * b + c$ (10 raised to the power 24 times a, plus 10 raised to the power 12 time b, plus c). Unfortunately, two calls to *rand* in the same statement will give the same result both times, so you need to choose values for a and b independently and then combine them. (Suppose you typed `m1 := 10^12*rand() + rand();`. Why wouldn't you ever find a prime testing values found this way?) Try finding two primes, each 36 digits long. The first can be found by starting with `m1:=4;` so you start out with a composite. Then use the following: `while (not isprime(m1)) do`

   ```
   a := rand();
   b := rand();
   c := rand();
   m1 := 10^24 * a + 10^12 * b + c;
   end do;.
   ```

   Multiply the two primes together and try to factor the result. If your machine is not fast enough, use the **STOP** button on the toolbar after a few minutes; the computation takes too long. Otherwise, create even longer primes and try factoring them. As you might imagine, no known algorithm would factor a 1024-bit (about 300 digits) number on your PC in your lifetime. It is easy to find the primes and multiply them together, but it is very difficult to factor the result! (Or RSA would not be secure.)

Try to get a feeling for how long it takes to factor numbers of different lengths; you can try changing the 10^{12} or 10^{24} in your *while loops* to larger or smaller values.

4. Find the multiplicative inverse of 25 modulo 43 (a number between 0 and 42, which when multiplied by 25 gives the result 1 modulo 43). You could try using `xmaple` and finding out about the function for computing the Extended Euclidean Algorithm by typing `?igcdex`. Does it help?

5. Try raising 5 to the power 19 modulo 21. (Use the algorithm for exponentiation given in class (or in your DM527 course) and on the last weekly note.)

6. Discuss problems 50 and 52 on page 611.

7. Discuss issues 2, 6 and 7 on pages 611–612.

8. Find four different square roots of 1 modulo 143 (numbers which multiplied by themselves modulo 143 give 1). Note that all of these numbers should be at least 0 and less than 143.

9. Add two of these different square roots which are not negatives of each other modulo 143 (two where adding them together does not give 143). Find the greatest common divisor of this result and 143. Subtract these same two different square roots and find the greatest common divisor of this result and 143. (Think about why you get these results.)

Assignment due 8:15, December 1

Late assignments will not be accepted. Working together is not allowed. (You may write this either in English or Danish.) Submit a single PDF file through the Blackboard system and include your name on the first page. If you submit this assignment more than once, use the same identification number both times. Remember to explain your answers (use comments).

1. Write a Bare Bones program to multiply a number in the variable X by 5.

2. Do one of the two following problems:

   (a) Suppose that between two asterisks on the Turing machine's tape, there is a decimal number, i.e., each cell contains a digit

   $$d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

   Design a Turing machine which starts on the rightmost asterisk and adds 15 to this number. Note that this might involve moving the leftmost asterisk to the left, and that the the original number could be larger than or smaller than 10.

   (b) Design a Turing machine that considers a string of zeros and ones between two asterisks as a non-negative integer in binary form and multiplies it by 5.