Institut for Matematik og Datalogi
Syddansk Universitet

November 6, 2009

Peter Schneider-Kamp

Introduction to Computer Science
E09 – Week 8

Announcement: Lab hours moved to Fridays 10-12

The lab hours for the second quarter are rescheduled from Thursdays 14-16 to Fridays 10-12. This change affects the labs in the Weeks 46, 48, and 50. If there are any problems with this rescheduling, please contact Peter Schneider-Kamp as soon as possible.

Announcement regarding Assignment due November 12

For the solution of the third task, you are advised to consult the excerpt on *Hamming Codes* available from the Course Documents folder in Blackboard.

Lecture: Monday, November 2

Tao Gu lectured on Networking, Internet, and error-correcting codes based on Chapters 4 and 1.9.

Lecture: Wednesday, November 4

Daniel Merkle lectured on Bioinformatics.

Lecture: Monday, November 9, 12-14 (U140)

Lone Borgersen will give an introduction to software engineering based on Chapter 7.

Lecture: Monday, November 16, 12-14

Jœrgen Bang-Jensen will start to lecture on the theory of computation based on Chapters 12.1 to 12.5.

Lecture: Wednesday, November 18, 14-16

Jœrgen Bang-Jensen will finish to lecture on the theory of computation and start lecturing on combinatorial optimization based on notes.

Discussion section: November 10, 10:15-12 (U37)

Consider the error-correcting code based on Hamming distance described in the document *Hamming Code* available from the Course Documents folder in Blackboard. Suppose we want to transmit messages of 7 bits.

1. How many redundant bits are required?
2. What will be the codeword transmitted by the sender if the message is 1101110?

The rest of the discussion section will deal with software engineering:

- Page 347, Problem 4.
- Page 351, Problems 1, 2, and 3.
- Page 354, Problem 1.
- Page 362, Problems 1, 3, and 4.
- Page 371, Problem 2.
- The library example distributed in class shows a little web application that can be used to search for books, to reserve books, to show status for borrowers, and to renew borrowing periods. Extend the distributed use case diagram to a library system that can also be used to borrow and return books. Make a description of the return process and discuss how changes of the use case diagram influence the other distributed diagrams.

(Danish original: Det udleverede biblioteksseksempel viser en lille netapplikation der kan benyttes til sgning efter bger, reservation af bger, visning af lnerstatus og til fornyelse af ln. Udbyg det udleverede brugsmnsterdiagram (use case diagram) til et bibliotekssystem, der ogs kan benyttes til udln og aflevering. Lav en beskrivelse af aflevering og diskuter hvordan ndringen af brugsmnsterdiagrammet (use case diagrammet) kommer til at pvirke de vrige udleverede diagrammer.)

- Page 375, Problems 2, 3, and 4.

Lab: November 13, 10:15-12 (terminal room above U49)

Discuss the following problems in groups of two or three.

First, you will be using the program `gpg` to try encryption. Usage information can be obtained by typing `gpg -h | more` (hitting the space bar will get the rest of it; the vertical line says pipe the output through the next program, and `more` shows a page at a time).

1. Create a public and private key using `gpg --gen-key`. You should choose DSA and El Gamal, and size 2048. Go to the directory `.gnupg` using `cd .gnupg`. List what is in the directory using `ls -al`. Try the commands `gpg --list-keys` and `gpg --fingerprint` to list the keys you have, with the fingerprints, which make it easier for you to check that you have the correct key from someone. How would you use fingerprints?

2. You can save your public key in a file in a form that can be seen on a screen using `gpg --export -a Your Name >filename`. You are "exporting" your key and specifying where the output should go. Then look at it using `more filename`; the `-a` made it possible to see it reasonably on your screen, since it changes it to ASCII.

3. Mail this file to someone else (either in another group or within your own group) or upload it to a public key server.

4. Try to figure out how to use `gpg` to "import" the public key you got from someone else. Check the fingerprint.

5. Create a little file and encrypt it. You can use `gpg -sea filename`. What does this do?

6. Mail your file to whoever has your public key. Read their file using the command `gpg -d inputfile >outputfile`. Then look at the output file you created.

7. You can also encrypt a file for your own use using a symmetric key system protected by a pass phrase. Try using `gpg --force-mdc -ca filename`. Then try decrypting as with the file you decrypted previously. Why might you want to do this?

8. 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`:

   (a) Small numbers.

   Start your Maple program, using the command `xmaple`. Type `restart;` at the beginning to make it easier to execute your work-

sheet after you have made changes. You can do this 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.)

(b) 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 ignore the warnings and join the execution groups through the `Edit` menu. 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.

(c) 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.

(d) 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?