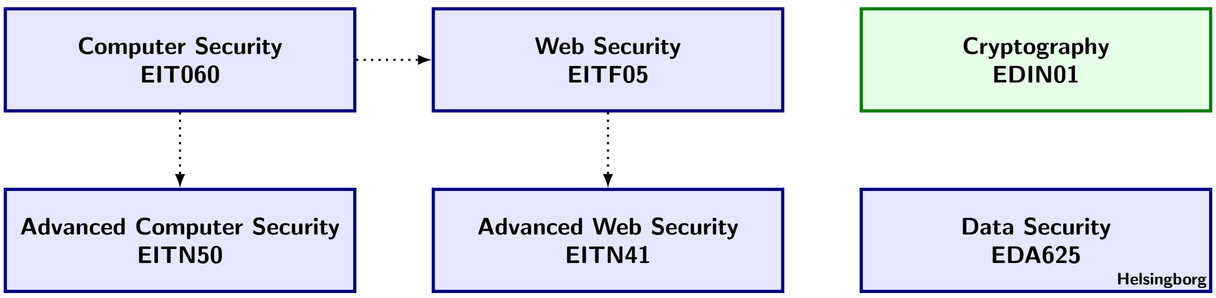
EDIN01 Cryptography 2015

Project 1: Factoring Algorithms

- This project will be done in groups of 1-2 people.

Learning goals:

- Understanding the complexity of factoring large numbers.
- Implementing a factoring algorithm.
- Improving presentation skills.



1 Introduction

The purpose of this project is to understand the difficulty, or more accurately the complexity, of factoring large numbers. We will only consider the case when our target number is a product of two large prime numbers. Such numbers are especially important in public key cryptography.

Let p and q be two prime numbers and let $N = pq$. Our factoring problem is given as follows.

Problem: Given N, find the primes p and q such that $N = pq$.

2 Trial division

We will first consider a very simple approach to the problem, usually referred to as *trial division*. This is based on the fact that if N is not a prime then N must be divisible by a number of size at most $\lfloor\sqrt{N}\rfloor$. So we just test whether

$$N \bmod p = 0,$$

for all $p = 2, 3, 4, \ldots, \lfloor\sqrt{N}\rfloor$.

Exercise 1: Assume that we can perform one million tests of the above kind each second on our computer. How long would it take to factor a 25 digit number (with two prime factors both of the same order (12 digit numbers))?

If you want to factor many 25 digit numbers, you can improve the running time of the basic trial division by first precomputing and storing the primes up to $\lfloor\sqrt{N}\rfloor$.

Exercise 2: How much faster does your improved trial division algorithm become for 25 digit numbers? Roughly how much storage does your algorithm require? What kind of budget does the storage requirement demand; student budget, big government grant, more dollars than there are atoms in the universe? You may check current storage pricing on `http://www.prisjakt.nu`.

3 More efficient methods

As you may guess, there are more powerful algorithms for factoring. Whereas the complexity of trial division is exponential ($N^{\beta} \cdot C$ for some constants β and C), modern factoring algorithms has a complexity that lies somewhere between polynomial and exponential time. Two well known methods are called *Quadratic Sieve* and *Number field Sieve*, respectively. The Quadratic Sieve algorithm was the fastest factoring algorithm up to early 1990s, but when the target numbers grew larger, the Number field Sieve started to outperform the Quadratic Sieve, due to a better asymptotic performance. Recently, we have seen several cases of 512 bit numbers (150 decimal digit numbers) being factored through a large computational effort. One conclusion is that using RSA with a 512 bit composite number is not secure enough. The minimum recommended length is currently 1024 bits.

Exercise 3: Write a program that implements a simple version of the *Quadratic Sieve algorithm*, described in the sequel. Use this program to factor the 25 digit number you have been given from the assistant.

As you will have understood from the previous exercise, efficient implementations are very important for the success of such an algorithm.

Exercise 4 (voluntary): Include measurement of CPU-time in your program and measure the time it takes to factor the number

$$n = 92434447339770015548544881401$$

Report the time to the assistant and compare your performance with other groups.

Reporting your results: Report your answers to the assistant. Show a printout of your program and explain how it works.

4 The Quadratic Sieve algorithm

The Quadratic Sieve algorithm is based on a very basic trick, known for a long time. *If* we in some way can find (or produce) two different numbers x and y such that

$$x^2 = y^2 \bmod N$$

then we have

$$x^2 - y^2 = (x-y)(x+y) = 0 \bmod N,$$

and this means that $(x-y)(x+y) = K \cdot p \cdot q$ for some integer K. We can separate in two possible cases,

- either p divides $(x-y)$ and q divides $(x+y)$, or vice versa;
- or both p and q divides $(x-y)$ and none of them divides $(x+y)$, or vice versa.

We calculate

$$d = \gcd(x-y, N).$$

In the first case we will then get $d = p$ or $d = q$ whereas in the second case we get $d = N$ or $d = 1$. If the first case appear we can factor N, but if the second case appears we can not.

Example: It is easy to check that $32^2 = 10^2 \bmod 77$. By calculating $d = \gcd(32-10, 77) = 11$ we get the factor 11 of 77.

In conclusion, given two integers x and y such that $x^2 = y^2 \bmod N$ we will be able to factor N with probability 1/2. Our current problem is how to obtain such numbers x, y.

In order to proceed we need some definitions. We define a *factorbase* F to be the set of prime numbers less than a certain bound B, called the *smoothness bound*. The number of primes in F is denoted $|F|$.

Example: If $B = 12$, then $F = \{2, 3, 5, 7, 11\}$.

Furthermore, we define a number x to be *B-smooth* if it can be factored over the factorbase F, i.e., it can be written as a product of primes all smaller than B.

Example: $N = 264$ is 12-smooth since $N = 264 = 2^3 \cdot 3 \cdot 11$.

The first step in Quadratic Sieve is to generate many numbers x_i, $i = 1, 2, \ldots$, such that if $x_i^2 = y_i \bmod N$ then y_i is a B-smooth number (y_i factors over F).

In order to find such numbers, we could select a random number r and test whether $r^2 \bmod N$ is B-smooth. But we can increase our probability of finding such numbers if we select the numbers r to test of a special form. We select our numbers r to test as

$$r = \lfloor \sqrt{k \cdot N} \rfloor + j, \tag{1}$$

for $j = 1, 2, \ldots$ and $k = 1, 2, \ldots$.

Note that

$$r^2 = (\lfloor \sqrt{k \cdot N} \rfloor + j)^2 = (\lfloor \sqrt{k \cdot N} \rfloor)^2 + 2 \cdot (\lfloor \sqrt{k \cdot N} \rfloor) \cdot j + j^2,$$

which means that $r^2 \bmod N$ is a number of the same order as $\sqrt{N}$. The probability that such a number will factor over F is much larger than for a randomly selected r.

Example: If $N = 77$ then $r = \lfloor \sqrt{1 \cdot N} \rfloor + 1 = 8 + 1 = 9$ and $r^2 \bmod 77 = 4$ is 12-smooth.

Remark: The name Quadratic Sieve comes from an efficient way of implementing the above step, using a sieving approach. However, we skip this idea here.

After this first step, we have a set of numbers x_i, $i = 1, 2, \ldots, L$, such that if $x_i^2 = y_i \bmod N$ then y_i factors over F. Now we are going to combine them to build two numbers x and y such that $x^2 = y^2 \bmod N$.

If we assume that $L > |F|$, say $L = |F| + 5$, then there exists a suitable selection of x_i's such that

$$x_{i_1}^2 \cdot x_{i_2}^2 \cdot \ldots x_{i_{L'}}^2 = y_{i_1} \cdot y_{i_2} \cdot \ldots y_{i_{L'}} \bmod N,$$

and $y_{i_1} \cdot y_{i_2} \cdot \ldots y_{i_{L'}} = Y^2$ for some Y.

Let us demonstrate why. The process is the same as Gaussian elimination for solving a system of linear equations.

Let the factor base F be $F = \{p_1, p_2, \ldots, p_{|F|}\}$. We write our collected numbers x_i as a system of equations

$$\begin{aligned} x_1^2 &= p_1^{e_{11}} \cdot p_2^{e_{12}} \cdot \ldots \cdot p_{|F|}^{e_{1|F|}} \\ x_2^2 &= p_1^{e_{21}} \cdot p_2^{e_{22}} \cdot \ldots \cdot p_{|F|}^{e_{2|F|}} \\ \vdots \quad & \quad \vdots \\ x_L^2 &= p_1^{e_{L1}} \cdot p_2^{e_{L2}} \cdot \ldots \cdot p_{|F|}^{e_{L|F|}}, \end{aligned}$$

where all numbers are assumed to be reduced modulo N. Our objective is to obtain an equation where the right hand side is of the form y^2 for some y. This is true if all exponents involved e_{ij}, $j = 1, \ldots |F|$, are *even*. If so, the right hand side

$$p_1^{e_{i1}} \cdot p_2^{e_{i2}} \cdot \ldots \cdot p_{|F|}^{e_{i|F|}} = \left(p_1^{e_{i1}/2} \cdot p_2^{e_{i2}/2} \cdot \ldots \cdot p_{|F|}^{e_{i|F|}/2}\right)^2$$

and this gives $y = p_1^{e_{i1}/2} \cdot p_2^{e_{i2}/2} \cdot \ldots \cdot p_{|F|}^{e_{i|F|}/2}$.

The remaining question is how to get the right hand side of one equation to have only even exponents. This is done by a Gaussian elimination process.

Start by selecting an equation for which the exponent for p_1 (e_{i1}) is odd, say

$$x_1^2 = p_1^{e_{11}} \cdot p_2^{e_{12}} \cdot \ldots \cdot p_{|F|}^{e_{1|F|}}.$$

Now go through all other equations. For each equation for which the exponent for p_1 (e_{i1}) is odd, say

$$x_i^2 = p_1^{e_{i1}} \cdot p_2^{e_{i2}} \cdot \ldots \cdot p_{|F|}^{e_{i|F|}},$$

we replace this equation by

$$(x_1 \cdot x_i)^2 = p_1^{e_{11}+e_{i1}} \cdot p_2^{e_{12}+e_{i2}} \cdot \ldots \cdot p_{|F|}^{e_{1|F|}+e_{i|F|}}.$$

If the exponent is even, we do nothing.

After this step, we remove the first equation (the one with the odd exponent) from the system. The exponents for p_1 are now even in all remaining equations. We now repeat the same process but for p_2, and so on. Note that once all exponents for p_1 are even they will remain even throughout the entire process.

After going through all the exponents for all the primes, we should, with $L = |F| + 5$, end up with 5 equations with all even exponents on the right hand side. We then have several pairs x, y such that $x^2 = y^2 \bmod N$, and this should give us a high probability of finding the factors of N.

As you might have noticed, the above procedure is equivalent to solving of a system of binary linear equations, where even exponent corresponds to the value 0 and odd exponent corresponds to the value 1.

An important note is the following. Let $r_1^2 = b_1$, where b_1 factors over F. Then the number $r = 2 \cdot r_1$ by $r^2 = (r_1 \cdot 2)^2 = 4 \cdot r_1^2 = b$ also produces a relation $r^2 = b$ where b factors over F. Note that for these two relations the binary row in the matrix above will be the same. Many pairs of relations of the above form will be created when you run the algorithm. It is clear that both of them represent the same relation and that only one of them should be included. *Therefore, before you insert a row in the binary matrix M above, make sure that it is not already present.*

Example: In this example we go through the factoring procedure, step by step. Make sure that you understand each step. Assume $N = 16637$ and we want to find the factorization of N. We take our factorbase as the first 10 prime numbers.

$$F = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29$$

Take $L = |F| + 2 = 12$. Hence, we want to find 12 relations such that $x_i^2 = y_i \bmod N$ can be completely factorized by the prime numbers in our factorbase, i.e., x_i^2 is *30-smooth*. By using (1) we can generate numbers that are likely to be *30-smooth*. Trying e.g., $k = 2$ and $j = 2$ will give $r = 184$ and $r^2 \bmod N = 582$. Since

$$582 = 2 \cdot 3 \cdot 97,$$

this can not be factored over our factorbase since $97 > 29$, so $r = 184$ is thrown away. Next, we try $k = 3$ and $j = 2$, which gives $r = 225$ and $r^2 \bmod N = 714$. This number is *30-smooth* since

$$714 = 2 \cdot 3 \cdot 7 \cdot 17$$

so it can be factorized by our factorbase. Hence, we let $x_1 = 225$. Continuing with different choices for k and j gives

k	j	r	$r^2 \bmod N$
3	2	225	$714 = 2 \cdot 3 \cdot 7 \cdot 17$
4	4	261	$1573 = 11^2 \cdot 13$
5	3	291	$1496 = 2^3 \cdot 11 \cdot 17$
5	4	292	$2079 = 3^3 \cdot 7 \cdot 11$
6	2	317	$667 = 23 \cdot 29$
7	2	343	$1190 = 2 \cdot 5 \cdot 7 \cdot 17$
10	6	413	$4199 = 13 \cdot 17 \cdot 19$
11	4	431	$2754 = 2 \cdot 3^4 \cdot 17$
12	12	458	$10120 = 2^3 \cdot 5 \cdot 11 \cdot 23$
13	4	469	$3680 = 2^5 \cdot 5 \cdot 23$
13	8	473	$7448 = 2^3 \cdot 7^2 \cdot 19$
14	8	490	$7182 = 2 \cdot 3^3 \cdot 7 \cdot 19$

Note that e.g., $r = 395$ gives $r^2 \bmod 16637 = 6292 = 2^2 \cdot 11^2 \cdot 13$. This also factors over our factorbase but it would give the same binary row in the matrix as $r = 261$. Because of this, $r = 395$ will also be thrown out.

Now we have 12 equations such that $x_i^2 = y_i \bmod N$ factors over the chosen factorbase. The binary matrix can now be written as

$$M = \begin{pmatrix}
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}$$

where each column represents one prime number in the factorbase and each row represents one of the 12 equations. By solving $\underline{x} \cdot M = \underline{0}$ we get a solution

$$\underline{x} = 101100000000.$$

This means that we multiply row 1, 3 and 4 together.

$$\begin{aligned} 225^2 \cdot 291^2 \cdot 292^2 &= 2^4 \cdot 3^4 \cdot 7^2 \cdot 11^2 \cdot 17^2 \mod 16637 \\ \Rightarrow (225 \cdot 291 \cdot 292)^2 &= (2^2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 17)^2 \mod 16637 \\ \Rightarrow 2787^2 &= 13850^2 \mod 16637 \end{aligned}$$

Now we calculate $\gcd(13850 - 2787, 16637) = 1$. Bad luck, we didn't get a factor. Another solution is

$$\underline{x} = 001000011100.$$

In the same way as before, we multiply row 3, 8, 9 and 10 in order to get the expression

$$4891^2 = 8952^2 \mod 16637.$$

Calculating $\gcd(8952 - 4891, 16637) = 131$ gives us a factor of N. Taking 16637/131=127 will give us the other factor.

A few words about implementation of the algorithm. The size of the factor base should for best performance in your case be as large as possible. Select a size that your computer can handle, for example $L = 1000$.

Finally, there are two things to address in your program. Firstly, you need to be able to do arithmetic with large numbers (30 digit numbers). The standard integer representation in a programming language is usually not sufficient. You need to use some library supporting large integers (or implement it yourself!). The second problem you might run into is memory management. Note that the second step of the algorithm requires a substantial amount of memory for large factor bases. You might need to check the efficiency of your representation in your program as well as how your programming language handles memory.

Here are some numbers you can use to test your program;

$323 = 17 \cdot 19,$
$307561 = 457 \cdot 673,$
$31741649 = 4621 \cdot 6869,$
$3205837387 = 46819 \cdot 68473,$
$392742364277 = 534571 \cdot 734687.$