FACTORING LARGE INTEGERS

R. Arigita Del Cacho

March 2003

R. Arigita Del Cacho

MA600 dissertation

Institute of Mathematics and Statistics
University of Kent at Canterbury

Abstract

When we were children, we were taught how to multiply two numbers. We were given multiplication tables for the numbers from 0 to 10, and then, big numbers were obtained by adding the result of the multiplication of smaller ones. What we were not taught was how to reverse the process to get those small numbers that were used to produce the big one. This is known as factorization. Factorization of integers is considered a NP complete problem and many great mathematicians have been working through this problem since ancient times. This document is an introduction, description and implementation of several integer factorization algorithms separated in two main categories, deterministics and probabilistics.

Acknowledgments

I'd like to gratefully thanks my family for its complete support while doing my degree. To my friends for being who they are. To those with whom I shared my time here. To those who taught me the mathematics discipline.

cold death came to knock my door
I shaken my head to spread the dust
through the darkness of the night
and the spoils of a wasted life
I prayed to stay alive

Preface

The topic covered in this paper is the building block, the basis, for many of the paths which can drive us to brake the ancient and difficult problem of factorization, in which some of the known to date public key encryption algorithms relay.

Chapter 1 introduces deterministic algorithms which has been heavily used for long time to factorise numbers up to some limit in its length, hand made easy to follow examples are provided for completion of the algorithms. Chapter 2 introduces probabilistic algorithms from more recent times, which are being used nowadays to factorise large integers. In Chapter 3 I try to express some observations about the previous factorization algorithms and try to introduce some information I gained while writing the paper. Appendix A gives a brief explanation about tools and formulaes shared by some of the algorithms.

During the period of writing this paper I went one by one implementing each of the algorithms into working programs to satisfy myself and anyone interested in computer programming, thus finally I decided to include the C++ source code of the algorithms in the Appendix B of the paper, Zero Day. Nowadays factorizations are performed in computational devices and therefore this programs are the perfect complement to the theory proposed here. Zero Day source code can be found in the CD-ROM accompanying the paper.

Notation

We use the letter N (except in Chapter 3 where we use K) to denote the number to be factorised, which is ussually compossed of only two prime divisors, e,g., $N = a \cdot b$. To factorise numbers with more than two prime divisors, the same algorithms can be applied recursively.

To know if an integer number is prime or composite we use an *Oracle* function which applies some primality testing procedure to the input number, and output a boolean if the number is or is not prime. Although factorization and primality testing are highly related topics, primality testing algorithms are beyond the purpose of this paper. There is an extense literature for primality testing algorithms, e.g. [2].

The *Big-O* notation: we say that $f(n) = O(g(n))$ if there exists a constant C such that $f(n)$ is always less than $C \cdot g(n)$, e.g., $23n^3 + 11^2 - 4 = O(n^3)$. It denotes the amount of effort, cycles, steps, bit operations needed to accomplish a task by a computation algorithm independently of which computational device it is implemented in. We use the Big-O notation to describe the asymptotic running time of the algorithms.

We use $|a|$ to denote the number of digits of a. The $\approx$ symbol to denote an equality approximation of the left and right hand side of the expression, e.g., $|a_1| \approx |a_2|$ meaning that the size of a_1 is approximately equal to the size of a_2.

We say that a number is *B-smooth* if it completely factorises into small primes bellow some upper bound B.

A *Factor Base* is a finite set of small prime numbers bellow some upper bound B.

We use an Oracle function to know if a prime number is a quadratic residue of N. Note that there are several algorithms for this purpose. See [2].

Most of the algorithms make use of the greatest common divisor, *gcd*, to find a common divisor of two integer numbers, this task is achieved using Euclid's algorithm, which can be found in [2, 4] or any other book of introduction to algebra together with congruences and modular arithmetic.

In Chapter 3 we introduce the set of "real numbers with arbitrary fixed length", $\mathbf{R}_A$, meaning a set of real numbers where we truncate at fixed A digits after the point, therefore casting to rational numbers, thus $\mathbf{R}_A \subset \mathbf{Q}$. We use $[a]_A$ to denote the real number a truncated to arbitrary fixed length A after the point.

Contents

Chapter 1

Deterministic Algorithms

time seems stopped now
and, it, keeps, re,pea,ting
will you please complete me?

1.1 Introduction

In this chapter I try to cover some effective techniques for factoring large composite integers using deterministic algorithms. These algorithms have a 100% chance of finding the factors of a given composite number, but their running time grows exponentially with the length of the number to be factorised. They are effective up to some limit in this length, say 20 to 30 digits, more than that is hopeless and resources consuming.

1.2 Trial Division Algorithm

This is the basic algorithm for finding the factors of an integer. It consists in making trial divisions of the number N to be factored by the primes below $\sqrt{N}$ until we completely factorise N into the product of small prime divisors.

There are two ways of implementing this algorithm on a computer program:

- Store a table of primes up to some limit and use a table lookup function to pick up each prime to be tested. This method is fast but there is a waste of storage space.
- Generate the primes up to the limit on running time. This leads to a slower factorization with the advantage that no storage is needed. Instead of trying each possible prime, which implies the use of some primality testing procedure, Riesel [1] suggest to test the small primes first, say 2, 3, 5, 7, and then all positive integers of the form $6k \pm 1$, for $k \in \mathbf{Z}$, $k \geq 2$ as trial divisors. This covers all the primes and some composite numbers, but the loop will be faster than generating only primes.

If we could have no limit in computer memory and big arrays of primes stored in tables, then the method of table lookup will be the preferred choice for an implementation of the

algorithm. There are big arrays of prime numbers distributed around the Internet, but as I mentioned before, the use of these arrays would involve a waste of memory and processor time spent in table lookups. Thus it is much easier to implement the second version of the algorithm, which runs very fast, and spend extra time dividing by some of the composite numbers generated by the sequence $6k \pm 1$ proposed.

1.2.1 Running Time

The worst case for this algorithm is when the prime factors are near the square root of N, thus the expected running time is about $O(N^{\frac{1}{2}})$.

1.2.2 Implementation

For an implementation of the Trial division algorithm I used the second method explained above. All numbers generated by the the sequence $6k \pm 1$ for $k \geq 2$ are tested against the number to be factorised, those with a remainder $== 0$ are returned as factors. Refer to Appendix B.2, for the source code of the algorithm.

1.3 Fermat's Factoring Method

The idea introduced by Fermat [1, 2, 4] is to write an odd composite number $N = a \cdot b$ as a difference between two square numbers.

$$N = a \cdot b = x^2 - y^2 = (x - y)(x + y) \tag{1}$$

therefore $x > \sqrt{N} > y$. We start computing $m = [\sqrt{N}] + 1$, which is the smallest possible value of x (unless N happen to be a square number), then consider $z = m^2 - N$ and check if this number is a square. If it is then we have found $N = x^2 - y^2$ and we are finished. Otherwise we try the next possible x, i.e. $x = m + 1$, compute $(m + 1)^2 - N = m^2 + 2m + 1 - N = z + 2m + 1$ and test whether this is square.

1.3.1 Example

$$N = 30361, \sqrt{N} = 174.244.., \; N \text{ is not a square.}$$

$$m = [\sqrt{N}] + 1 = 175, \; z = 175^2 - 30361 = 264$$

$z = 264$ is not a square so we start the iteration:

m	$2m+1$	z	is square?
175	351	264	no
176	353	615	no
177	355	968	no
178	357	1323	no
179	359	1680	no
180	361	2039	no
181	363	2400	no
182	365	2763	no
⋮	⋮	⋮	⋮
199	399	9240	no
200	401	9639	no
201	403	10040	no
202	405	10443	no
203	407	10848	no
204	409	11255	no
205	411	11664	yes!

In the last line we have $z = 11664 = 108^2$, so we stop the iteration. From this and (1) we can find the factorisation of N: put $x = m$, $y = \sqrt{z}$ in (1), then

$$\begin{aligned} N &= 30361 \\ &= 205^2 - 108^2 \\ &= (205 - 108)(205 + 108) \\ &= 97 \cdot 313 \end{aligned}$$

and we found the prime factors 97 and 313 of $N = 30361$.

1.3.2 How much work is required?

If $N = a \cdot b$ with $a < b$ (the usual case) then the factorization will be achieved when m reaches $x = \frac{(a+b)}{2}$. Since the starting value of $m \approx \sqrt{N}$, and $b = \frac{N}{a}$, this will take

$$\approx \frac{1}{2}(a + \frac{N}{a}) - \sqrt{N} = \frac{(\sqrt{N} - a)^2}{2a} \tag{2}$$

iterations. Following our previous example, we can apply (2) to verify the approximate number of iterations we used to find the factors 97 and 313:

$$\frac{(174.244 - 97)^2}{2 \cdot 97} = 30.75 \approx 31$$

The reader can refer to the table above and count its rows or subtract the final minus the initial value of m to verify this approximation.

Fermat's method is practical when the two factors are very close to $\sqrt{N}$, otherwise the amount of iterations needed gets very large. Consider the situation where

$$N = a \cdot b,\ a \approx N^{\frac{1}{3}},\ b \approx N^{\frac{2}{3}}$$

then the number of cycles will be

$$\frac{(\sqrt{N} - \sqrt[3]{N})^2}{2\sqrt[3]{N}} = \frac{(\sqrt[3]{N})^2(\sqrt[6]{N} - 1)^2}{2\sqrt[3]{N}} \approx \frac{1}{2}N^{\frac{2}{3}}$$

which is higher than $O(N^{\frac{1}{2}})$ and therefore impractical.

1.3.3 Implementation

See Appendix B.3, for the source code of the algorithm.

1.4 Legendre's Contribution

Legendre contributed substantially [1, 2, 4] to the development of factorization algorithms introducing a procedure for finding small *quadratic residues*[1] of N. This procedure is known as the *continued fraction expansion*[2] of $\sqrt{N}$.

Another two important contributions by Legendre was the introduction of its quadratic congruence, which will be the goal to achieve by many of the factorization algorithms that came later in time, and the Legendre's Symbol.

1.4.1 Legendre's Congruence

Consider an integer $N = a \cdot b$, a and b primes, then the congruence

$$x^2 \equiv y^2 \pmod{N} \tag{3}$$

has four solution which can be used to factor N, ie:

$$u^2 \equiv y^2 \pmod{a} \text{ has two solutions}$$

$$u \equiv \pm y \pmod{a}$$

and

$$v^2 \equiv y^2 \pmod{b} \text{ has two solutions}$$

$$v \equiv \pm y \pmod{b}$$

[1] See Appendix A.1
[2] See Appendix A.2

Thus the congruence $x^2 \equiv y^2 \pmod{ab}$ has four solutions. To factor N we have to find a non trivial solution of (3). Since $x^2 - y^2 = (x+y)(x-y) \equiv 0 \pmod{N}$ and $x+y$ or $x-y$ is not divisible by both a and b, then either $x+y$ or $x-y$ must be divisible by a and the other by b. The factor a (or b) can be extracted applying Euclid's algorithm on $x+y$ (or $x-y$) and N, ie: $\gcd(x \pm y, N)$

Several methods make use of Legendre's congruence to find the factors of N, the difference between them is the way in which the solution to (3) is found.

1.4.2 Legendre's Symbol

If a is a quadratic residue $\pmod{p}$, p prime, then the symbol (a/p) is given the value $+1$, and -1 if a is a quadratic non residue.

1.5 Shank's Factoring Method (SQUFOF)

Shank's method [1, 4] computes the regular fraction expansion of $\sqrt{N}$ introduced by Legendre, using (18) and (19) from Appendix A.2 until a square denominator Q_n that has never previously occurred during the computation is found in an even number of steps. This will guarantee a non trivial solution to Legendre's congruence (3) as

$$A_{n-1}^2 \equiv (-1)^n Q_n \equiv R^2 \pmod{N} \tag{4}$$

where

$$A_s = b_s A_{s-1} + A_{s-2} \pmod{N} \tag{5}$$

Then, the factors a and b of N can be obtained by means of Euclid's algorithm applied to $A_{n-1} \pm R$ and N, i.e., $\gcd(A_{n-1} \pm R, N)$.

1.5.1 Example

Find the factors of $N = 1436453$, set $b_0 = x_0 = \left\lfloor \sqrt{N} \right\rfloor = 1198$ then

$$\begin{aligned}
x_1 &= \frac{1}{\sqrt{N} - 1198} = \frac{\sqrt{N} + 1198}{1249} = 1 + \frac{\sqrt{N} - 51}{1249} \\
x_2 &= \frac{1249}{\sqrt{N} - 51} = \frac{\sqrt{N} + 51}{1148} = 1 + \frac{\sqrt{N} - 1097}{1148} \\
x_3 &= \frac{1148}{\sqrt{N} - 1097} = \frac{\sqrt{N} + 1097}{873} = 11 + \frac{\sqrt{N} - 1136}{203} \\
x_4 &= \frac{203}{\sqrt{N} - 1136} = \frac{\sqrt{N} + 1136}{719} = 3 + \frac{\sqrt{N} - 1021}{719} \\
&\vdots
\end{aligned}$$

$$\begin{aligned} x_{24} &= \frac{203}{\sqrt{N}-1010} = \frac{\sqrt{N}+1010}{2051} = 1+\frac{\sqrt{N}-1041}{2051} \\ x_{25} &= \frac{2051}{\sqrt{N}-1041} = \frac{\sqrt{N}+1041}{172} = 13+\frac{\sqrt{N}-1195}{172} \\ x_{26} &= \frac{172}{\sqrt{N}-1195} = \frac{\sqrt{N}+1195}{49} \end{aligned}$$

$Q_{26} = 49 = 7^2$ is a square when $n = 26$ is even. Now we have to compute

$$A_{25} \equiv Q_{26} = 7^2 \pmod{N}$$

to get the factors of N. This task can be achieved using (5) with initial values $A_{-1} = 1,\ A_0 = b_0 = \left\lfloor \sqrt{N} \right\rfloor = 1198,\ N = 1436453.$

$$\begin{aligned} A_1 &= 1 \cdot 1198 + 1 \equiv 1199 \\ A_2 &= 1 \cdot 1199 + 1198 \equiv 2397 \\ A_3 &= 11 \cdot 2397 + 1199 \equiv 27566 \\ &\vdots \\ A_{25} &= 13 \cdot 1247117 + 1179329 \equiv 154414 \end{aligned}$$

Put $x = 154414$ and $y = 7$ in Lengendre's congruence (3) and solve it applying Euclid's algorithm, i.e.:

$$\begin{aligned} \gcd(154414 - 7, N) &= 4679 \\ \gcd(154414 + 7, N) &= 307 \end{aligned}$$

therefore we found a factorization of $N = 4679 \cdot 307$.

Shank's avoids the computation of $A_n \pmod{N}$ by means of (5), instead he calculates a new fraction expansion by altering a sign in the numerator and taking the square root of the denominator in the last step of the iteration, and expands this new fraction until the coefficients in the numerator of two consecutive steps are equal. At this point, if the denominator Q_n of the last equation is odd, then it will be prime and a factor of N, if it is even then $Q_n/2$ will be prime and a factor of N. Thus continuing our example we have:

$$\frac{\sqrt{N}+1195}{49} \longmapsto \frac{\sqrt{N}-1195}{7}$$

$$\begin{aligned} q_1 &= \frac{7}{\sqrt{N}-1195} = \frac{\sqrt{N}+1195}{1204} = 1+\frac{\sqrt{N}-9}{1204} \\ q_2 &= \frac{1204}{\sqrt{N}-9} = \frac{\sqrt{N}+9}{1193} = 1+\frac{\sqrt{N}-1184}{1193} \\ &\vdots \end{aligned}$$

$$
\begin{aligned}
q_{11} &= \frac{233}{\sqrt{N}-995} = \frac{\sqrt{N}+995}{1916} = 1+\frac{\sqrt{N}-921}{1916} \\
q_{12} &= \frac{1916}{\sqrt{N}-921} = \frac{\sqrt{N}+921}{307} = 6+\frac{\sqrt{N}-921}{307}
\end{aligned}
$$

In the last two steps we have the same numerator, $P_{11} = P_{12} = 921$, so we stop. The denominator $Q_{12} = 307$ is prime and $N/307 = 4679$, so we have a factorization of $N = 307 \cdot 4679$.

The advantage of Shank's method for finding the factors after we found the square denominator Q_n in the first part of the algorithm, lies in that it is much faster to compute the fraction expansion of the new transformed equation than to compute the iterative recursion (5) to get the value of A_{n-1} that will be used in (4) to solve the congruence.

1.5.2 Resume of the Algorithm

Let

$$
P_0 = 0,\ Q_0 = 1,\ Q_1 = N - P_1^2,\ q_i = \left[\frac{\sqrt{N}+P_i}{Q_i}\right],
$$

$$
P_{i+1} = q_i Q_i - P_i,\ Q_{i+1} = Q_{i-1} + (P_i - P_{i+1})q_i
$$

If any Q_i is $< 2\sqrt{2\sqrt{N}}$ it is stored in a list which will contain the numbers useless for the factorization. Continue the iteration until some $Q_{2i} = R^2$ and compare it with the numbers in the list. If R (or $R/2$ if R is even) is included in the list then continue, else R (or $R/2$) is a valid square. Now the expression is:

$$
q_{2i} = \left[\frac{\sqrt{N}+P_{2i}}{Q_{2i}}\right] = \left[\frac{\sqrt{N}+P_{2i}}{R^2}\right]
$$

Continue the expansion of $\frac{\sqrt{N}-P_{2i}}{R}$ in the same way as before until some $P_{i+1} = P_i$, which will occur approximately after half of the step than it took to find $Q_{2i} = R^2$. Then, finally Q_i or $Q_i/2$ if Q_i is even will be a factor of N.

1.5.3 Running Time

Experience points to an expected running time of $O(\sqrt[4]{N})$, however it has the advantage, over some other probabilistic algorithms, of performing simple arithmetic operations in each cycle.

1.5.4 Implementation

See Appendix B.4, for the source code of the algorithm.

1.6 Morrison and Brillhart Continued Fraction Method

Known as CFRAC algorithm [6, 1, 2, 4], the technique used is similar to that of Shank's algorithm. Find a non trivial solution to the Legendre's congruence (3) and then obtain a factor of N by means of Euclid's algorithm applied to $(x \pm y, N)$. Two ideas are used to solve the congruence (3), one is to find small quadratic residues from the continued fraction expansion of $\sqrt{N}$, and the other due to Maurice Kraïtchik in which known residues are combined to form new ones, in this case squares.

1.6.1 Description of the algorithm

Morrison and Brillhart look for a combination of the quadratic residues generated by the fraction expansion of $\sqrt{N}$ to yield a square, and thereby use fewer cycles than Shank's method. Compute part of the fraction expansion of $\sqrt{N}$ using (18,19), in the search for a solution of (4). While Shank's method requires the quantities P_n and Q_n only, this method requires also $A_{n-1} \pmod N$, (5).

We set an upper bound U_b and use the first B primes with Legendre's symbol $(N/p_i) = 1$ to form what is called a *factor base*, then the Q_i's are searched for prime factors $\leq U_b$. Only those B-smooth Q_i's (which factor completely within the factor base) are retained. The time consuming part of this algorithm lies in the search of sufficient number of Q_i's completely factored within the factor base. The factorizations discovered are stored as binary vectors $\vec{V} = (v_0, v_1, ..., v_B)$ where

$$v_i = \begin{cases} 0 & \text{if the exponent of } p_i \text{ is even} \\ 1 & \text{if the exponent of } p_i \text{ is odd} \end{cases}$$

The next step is to search for square combinations of the completely factored Q_i's. To achieve this combinations first generate $B + 1$ complete prime factorizations and perform Gaussian elimination $\pmod 2$ on the $(B + 1) \times B$ matrix whose rows are the vectors $\vec{V_i}$ of each factored Q_i. As soon as a linear combination of vectors in the matrix yields a vector with all entries equal to 0, there is a 50% chance that the combination of Q_i's, which is now a square, will lead a factorization of N. If not we perform subsequent linear combinations of the $\vec{V_i}$'s to get more linearly dependent vectors which we can use to find a non trivial solution of Legendre's congruence (3) and use Euclid's algorithm to get a factor of N.

1.6.2 Example

Find the factors of $N = 1436453$.

We first choose a factor base comprising all primes below $U_b = 50$ with Legendre symbol $(N/p_i) = 1$ for all $p_i < U_b$, and include -1 as one of the factors for those Q_n's with n odd. Thus our factor base $= \{-1, 7, 17, 29, 41\}$. Using the fraction expansion of the example in

section 1.5.1. we get

n	$(-1)^n Q_n$	-1	7	17	29	41
3	$-1 \cdot 7 \cdot 29$	1	1	0	1	0
6	$7 \cdot 17$	0	1	1	0	0
11	$-1 \cdot 17 \cdot 29$	1	0	1	1	0
18	29^2	0	0	0	0	0
23	$-1 \cdot 7 \cdot 29$	1	1	0	1	0
26	7^2	0	0	0	0	0

We have few Q_n's completely factored into primes of our factor base, and the matrix associated to the exponents of each of the factors. Before proceeding further with the algorithm and perform gaussian elimination over the exponent matrix, augmented with an identity matrix to keep track of the operations needed to reduce it, we can see that two of the rows in the factorization table are squares, therefore we may attempt a factorization using (4) as follows,

$$A_{17}^2 \equiv Q_{18} = 29^2$$

using (5) with initial values as in Example 1.5.1. we find

$$A_{17} \equiv 1207153 \equiv 29^2 \pmod{N}$$

so we try Euclid's algorithm

$$\gcd(1207153 - 29, N) = 307$$

which yields a factor of N, and $N = 307 \cdot 4679$

Note that in this example we had a Q_n which factored into primes of the factor base and itself was a square, so we didn't need to find a dependent vector amongst the ones in the exponent matrix, thus reducing our search to just few cycles. But for bigger numbers it will not be this case and gauss elimination will be needed to find a combination of Q_ns which is a square, to then use (4) and *gcd* to find a non trivial factor of N.

1.6.3 Running Time

Practical implementations of the algorithm points to an expected running time of

$$C \cdot N^{\sqrt{1.5 \ln \ln N / \ln N}}$$

Chapter 2

Probabilistic Algorithms

tried to find myself but myself keeps sleeping away
all I've done for the ones that are allowed to stay
tried to sieve myself but myself keeps sleeping away

2.1 Introduction

More recently, mathematicians such as Pollard or Pomerance introduced random values into new algorithms in the search for a solution of Legendre's Congruence (3). This algorithms proved to be very successful and has been used to factor some of the largets composite integers factorised to date. Later, Lenstra introduced elliptic curves into factorization, which based on similar ideas to the ones of Pollard's, make use of elliptic curves properties and arithmetic to find a non trivial factorization of any composite integer.

2.2 Pollard's ρ Method

Pollard's ρ method [14, 15, 1, 2, 4] is based on a cycle finding algorithm. Let N be a composite number and p an unknown non trivial divisor of N. Let $f(x)$ be a simple irreducible polynomial in x (for practical purposes we can use $f(x) = x^2 - a$ or something similar, where a is constant not 0 or -2). Starting with an integer x_0, we create a sequence from the recursive definition

$$x_i = f(x_{i-1}) \pmod{N}$$

e.g., let $N = 1651$, $x_0 = 2$, $f(x) = x^2 + 3$, then our sequence x_i will be:

$$\begin{aligned} x_0 &= 2 \\ x_1 &= 7 \\ x_2 &= 52 \\ x_3 &= 1056 \\ x_4 &= 714 \\ x_5 &= 1291 \end{aligned}$$

Let $y_i = x_i \pmod{p}$. In our example we choose $p = 13$, then the sequence y_i is:

$$\begin{aligned} y_0 &= 2 \\ y_1 &= 7 \\ y_2 &= 0 \\ y_3 &= 3 \\ y_4 &= 12 \\ y_5 &= 4 \end{aligned}$$

Since $x_i \equiv f(x_{i-1}) \pmod{N} \Rightarrow y_i \equiv f(y_{i-1}) \pmod{p}$, there are only a finite number of congruence classes $\pmod{p}$ (namely p of them), and so we will have $y_i = y_j$ for some pair (i, j), but when that happens, we will keep cycling, and for $t = 1, 2, 3...$,

$$y_{i+t} = y_{j+t}$$

The sequence of y_i's look like a circle with a tail, similar to the greek letter ρ.

If $y_i = y_j$, then $x_i \equiv x_j \pmod{p}$ and so $p \,|\, (x_i - x_j)$. If $x_i \neq x_j$ then $\gcd(x_i - x_j, N)$ is a factor of N. As we do not know p, we do not know when $y_i = y_j$, but if the length of the cycle is l, then once we are off the tail of the sequence, any pair (i, j) for which $l \,|\, (j - i)$ will work. We find some systematic way of choosing a lot of pairs (i, j) and for each such pair compute $\gcd(x_i - x_j, N)$, if it is $\neq 1$ or N then it is a factor of N.

2.2.1 Example

Factor $N = 337423$ with $f(x_i) = x_{i-1}^2 + 3$, $x_0 = 2$

$$\begin{aligned} x_1 &\equiv 7 \\ x_2 &\equiv 52 \\ y_1 &= x_2 - x_1 \equiv 45,\ gcd(45, N) = 1 \\ x_3 &\equiv 2707 \\ x_4 &\equiv 241969 \\ y_2 &= x_4 - x_2 \equiv 241917,\ gcd(241917, N) = 1 \\ x_5 &\equiv 32850 \\ x_6 &\equiv 43749 \\ y_3 &= x_6 - x_3 \equiv 41042,\ gcd(241917, N) = 1 \end{aligned}$$

$$
\begin{aligned}
x_7 &\equiv 111748 \\
x_8 &\equiv 265123 \\
y_4 &= x_8 - x_4 \equiv 23154,\ gcd(23154, N) = 1 \\
x_9 &\equiv 270310 \\
x_{10} &\equiv 232568 \\
y_5 &= x_{10} - x_5 \equiv 199718,\ gcd(199718, N) = 1 \\
x_{11} &\equiv 317419 \\
x_{12} &\equiv 313764 \\
y_6 &= x_{12} - x_6 \equiv 270015,\ gcd(270015, N) = 383
\end{aligned}
$$

We found one factor of N and $N/383 = 881$ is the other, so $N = 383 \cdot 881$.

2.2.2 Heuristics for Pollard's ρ Method

The factor y_i included in the product Q_i for the computation of (Q_i, N) contains at least $i + 1$ algebraic factors:

$$
\begin{aligned}
y_i &= x_{2i} - x_i = x_{2i-1}^2 + a - (x_{i-1}^2 + a) = x_{2i-1}^2 - x_{i-1}^2 \\
&= (x_{2i-1} + x_{i-1})(x_{2i-1} - x_{i-1}) \\
&\vdots \\
&= (x_{2i-1} + x_{i-1})(x_{2i-2} + x_{i-2}) \cdots (x_i + x_0)(x_i - x_0).
\end{aligned}
$$

The numbers x_i grow very fast, their number of digits is doubled from one x_i to the next because a squaring is performed in the recursion formula $x_{i+1} = x_i^2 + a$, hence x_i will be of order of magnitude $x_0^{2^i}$. So the expected number of prime factors $\leq G$ of y_i (which is a product of $i + 1$ large numbers), is $\approx (i + 1) \ln \ln G$. For n cycles of the algorithm we accumulate in Q_n the primes of all the factors $y_1, y_2, ..., y_n$, which together can be expected to include

$$\ln \ln G \sum_{i=1}^{n} (i + 1) \approx \frac{1}{2} n^2 \ln \ln G$$

prime factors $\leq G$. The number of primes below p is $\pi(p) \approx p / \ln p$, and therefore n ought to be so large that by choosing a suitable scale factor C, $C \cdot 0.5 n^2 \ln \ln p$ attains the magnitude of $\pi(p)$:

$$C \cdot \frac{1}{2} \ln \ln p \approx \frac{p}{\ln p}$$

i.e.

$$n = const. \times \left(\frac{p}{\ln p \ln \ln p} \right)^{\frac{1}{2}}$$

Thus the number of cycles needed to find a factor p of N is slightly smaller than $C\sqrt{p}$, and the expected running time about $O(\sqrt{p})$.

2.2.3 Implementation

Source code for the algorithm can be found in Appendix B.6. Small modifications to the program code, such as change of polynomial or constant term, can improve personal factorizations.

2.3 Pollard's $p-1$ Method

Pollard's $p-1$ method [1, 2, 3, 10, 11] assumes that the number N to be factored has a prime factor p with the property that $p-1$ factorises into small primes, say less than 100000, then $p-1 \,|\, 100000!$. Since exponentiation MOD N is fast, we can compute

$$m = 2^{100000!} \pmod{N}$$

quickly:

$$2^{100000!} = ((((2^1)^2)^3)^4...)^{100000}$$

By Fermat's theorem ($a^{p-1} \equiv 1 \pmod{p}$), since $p-1 \,|\, 100000!$, m is congruent to 1 (mod p), so $p \,|\, m-1$. Again there is a chance that N does not divide $m-1$, so that $\gcd(m-1, N)$ will be a factor of N. Note that we can use any number b relatively prime to N instead of 2 to compute m, and the same observations will hold.

In practice we must check the value of $\gcd(b^{k!}-1, N)$ periodically to find out if we have picked up the first prime divisor of N. If it is 1 we continue, if it is N we have picked up all factors of N and we have to go back or try another value for b. If $\gcd(b^{k!}-1, N) \neq 1 \, or \, N$ then it is a factor of N. To speed up the process we can pre-compute a list of primes and primes powers up to some limit, say 100000 as before. For each prime power write its correspondent prime instead, next choose an initial value for a, e.g. $a = 11$ and compute

$$a_{i+1} \equiv a_i^{p_i} \pmod{N}$$

where p_i is the i^{th} integer in the list of primes and prime powers. Then compute the accumulated product

$$Q_n \equiv \prod_{i=1}^{n} (a_i - 1) \pmod{N}$$

and check (Q_n, N) periodically to see if a factor of N has emerged, e.g. at intervals of 20 cycles.

2.3.1 Example

$N = 2323$, prime powers list max = 100, initial value $a_1 = 3$, check $\gcd(Q_n, N)$ each 10 cycles.

$$\begin{aligned} a_1 &= 3^1 \equiv 3 \\ a_2 &= 3^2 \equiv 9 \\ a_3 &= 9^3 \equiv 729 \\ a_4 &= 729^2 \equiv 1797 \\ a_5 &= 1797^5 \equiv 36 \\ a_6 &= 36^6 \equiv 1248 \\ a_7 &= 1248^7 \equiv 1498 \\ a_8 &= 1498^2 \equiv 2309 \\ a_9 &= 2309^3 \equiv 1902 \\ a_{10} &= 1902^{10} \equiv 910 \end{aligned}$$

$$\begin{aligned} Q_{10} &\equiv 2 \cdot 8 \cdot 728 \cdot 1796 \cdot 35 \cdot 1247 \cdot 1497 \cdot 2308 \cdot 1901 \cdot 909 \pmod{N} \\ &\equiv 1515 \end{aligned}$$

$$\gcd(1515, N) = 101$$

and we found a factorization of $N = 2323 = 101 \cdot 23$.

Note if the factor 101 would not have arise in the *gcd* operation, we should have continued generating a_i's until $i = 20$ and then checked the $\gcd(Q_{20}, N)$ in the search for a factor. In this example the numbers were so small that it worked in only ten cycles of computations of a_i's, and one only *gcd* operation.

2.3.2 Implementation

See Appendix B.7 for the implementation of the algorithm into C++ source code.

2.4 Carl Pomerance's Quadratic Sieve

Before going into the details of the Quadratic Sieve [1, 2, 3, 7, 16, 17], I shall briefly explain a previous algorithm due to J.D. Dixon [2] which based on similar ideas than CFRAC, introduced a different procedure to find Q_n's which completely factorised within a precomputed factor base to yield a solution of Legendre's congruence (3).

2.4.1 Dixon's ideas and algorithm

Dixon's idea was to choose a random integer r and compute the polynomial

$$g(r) = r^2 \pmod{N}$$

where N is the number to be factored, then factor $g(r)$ up to a limit, say B, with trial division to find all small prime divisors. If $g(r)$ does not factorise within this factor base with primes $< B$, discard it and pick up another r. The point is to accumulate more $g(r)$'s that are completely factored, than primes in the factor base.

Let $p_1, p_2, ..., p_B$ be the first B primes. If $g(r)$ factor completely then we can write it as

$$g(r) = p_1^{a_1} \times p_2^{a_2} \times \cdots \times p_B^{a_B} ,$$

as in CFRAC, we record the exponents of the factors of each $g(r)$ in a binary vector, $\vec{V}$, and as we have more vectors than primes in the factor base we can find a combination of them with all entries equal to 0 by means of Gaussian elimination over the exponent matrix, which implies that the combination of $g(r)$'s that produced the $\vec{0}$ vector is a square. Then

$$g(r_1) \times g(r_2) \times \cdots \times g(r_t) \equiv r_1^2 \times r_2^2 \times \cdots r_t^2 \pmod{N}$$

and we can attempt to solve (3) using Euclid's algorithm with a $50 - 50$ chance that it will yield a factor of N.

Computing the $g(r)$ polynomial is very quick, reducing a big $(B + EXTRA) \times (B)$ matrix is also very quick, but finding enough r_i's for which $g(r_i)$ completely factorises over the factor base is very slow and time consuming, and here is where the improvements of Carl Pomerance come into play.

2.4.2 Pomerance's Improvements

Pomerance's [3] main idea was to incorporate a sieving procedure to the above algorithm which enables us to perform trial division over a big range of numbers without doing any division at all, thus finding all the $g(r)$'s needed to form the exponent matrix much quicker than proving trial division on each $g(r)$ separately. The theory works as follows:

Instead of using $g(r) = r^2 \pmod{N}$, Pomerance employ the polynomial

$$f(x) = ([\sqrt{N}] + x)^2 - N \tag{6}$$

where if x is an integer, then

$$([\sqrt{N}] + x)^2 \equiv f(x) \pmod{N} \tag{7}$$

is a non trivial congruence, i.e., it is not an equality. Suppose we find a set of distinct integers $x_1, ..., x_k$ such that $f(x_1) \cdots f(x_k)$ is a square, say,

$$f(x_1) \cdots f(x_k) = Y^2$$

then let

$$X = ([\sqrt{N}] + x_1) \cdots ([\sqrt{N}] + x_k)$$

From (7) we see that $X^2 \equiv Y^2 \pmod{N}$, so we have a solution of Legendre's congruence (3). Now we still haven't solve the problem of finding a set $x_1, ..., x_k$ such that $f(x_1) \cdots f(x_k)$ is a square. For this purpose we build again a factor base B containing the first B small primes with Legendre symbol $(N/p_i) = 1$ for all $p_i \in B$, so each p_i in the factor base is a quadratic residue $\pmod{N}$. Then we compute the two roots of the quadratic congruence

$$N \equiv r^2 \pmod{p_i}$$

for all $p_i \in B$ using *Shanks-Tonelli* algorithm[1], call them a_{1_p}, a_{2_p}. We define a sieving range R as a closed interval crossing $\sqrt{N}$, say $[[\sqrt{N}] - M, [\sqrt{N}] + M]$, where M is suitably chosen according to the length of the number to be factored. For each x in our predefined range R of consecutive x's, compute $[\log_2 f(x)]$ (the number of binary bits in $f(x)$) and store these in locations indexed by x. Then for each of our primes p_i in the factor base we subtract $[\log_2 p_i]$ from the number in location x if and only if

$$x \equiv a_{1_{p_i}} \text{ or } a_{2_{p_i}} \pmod{p_i} \tag{8}$$

Once we found an x such that (8) holds, then we know that p_i will also divide $f(x + p_i)$, $f(x + 2p_i)$, ... , so we subtract $[\log_2 p_i]$ from all those entries indexed by $x + p_i$, $x + 2p_i$, etc, included in our sieving range R. At the end of the sieving, those entries in the array of $[\log_2 f(x_i)]$ for all $x_i \in R$ which have completely factorised, should be equal to 0, but this is not the case in practice because we do not sieve the possible powers of the primes which divide $f(x_i)$, so we allow for an error, and we look for those entries in the array that are below some Threshold value.

The sieve works for a simple reason based on arithmetic of logarithms: if p divides $f(x)$ then we subtract $\log_2 p$ from $\log_2 f(x)$ to get the remaining unfactorised part of $f(x)$, and continue subtracting $\log_2 p$ for all p's dividing $f(x)$ from the actual $\log_2 f(x)$ until we get to 0, which should happen if we subtract all possible powers of the primes which divide $f(x)$, and no round off error were introduced by computer arithmetic.

After the sieving is done, we perform trial division over those $f(x_i)$'s for which the value $[\log_2 f(x_i)]$ is below our predefined Threshold; if it completely factorised we store it in an array together with its correspondent x_i and build a binary vector $\vec{V}(x_i)$ of length equal

[1] See Appendix A.3

to the length of the factor base, $| \vec{V}(x_i) | = B$, which entries are the exponents $\pmod 2$ of the primes that factorised $f(x_i)$. We need more $f(x_i)$ completely factorised than factors in the factor base, say $B + \text{EXTRA}$, so

$$f(x_i) = \prod_{j=1}^{B} p_j^{v_{ji}}, \quad 1 \leq i \leq B + \text{EXTRA}$$

and for each i

$$\vec{V}(x_i) = (v_{1i}, v_{2i}, \ldots, v_{B_i}) \pmod 2, \quad 1 \leq i \leq B + \text{EXTRA}$$

With the binary vectors we build a $(B + \text{EXTRA}) \times (B)$ matrix and augment it with an identity matrix of size $(B + \text{EXTRA}) \times (B + \text{EXTRA})$. Then we perform Gaussian elimination to obtain EXTRA binary vectors linearly dependents, so it's entries will all be 0,

$$\vec{V}(x_i) = \vec{0}, \quad B < i \leq B + \text{EXTRA}$$

This insures us that the combination of $f(x)$ that produced these vectors is a square. To find such $f(x)$'s we look up the resulted vector of the identity matrix corresponding to the $\vec{V}(x_i) = \vec{0}$, and for each entry $= 1$ we pick up the corresponding $f(x)$ in the same position from the array of completely factorised $f(x)$'s. With all them we form a product to get Y^2, and with the x's in the same position we form another product to get X. Now we are ready to solve Legendre's congruence (3) with

$$\begin{aligned} X &= \prod x_{ij} \pmod N \\ Y^2 &= \prod f(x_{ij}) \\ Y &= \sqrt{Y^2} \pmod N \end{aligned}$$

$$0 \leq i < B + \text{EXTRA}, \quad B < j < B + \text{EXTRA}, \quad v_i = 1 \in \vec{V}(id_j)$$

and get the factors applying the $\gcd(X - Y, N)$.

2.4.3 Optimal Values and Running Time

The Quadratic Sieve algorithm requires some optimal values for two important variables,

- the factor base length (B),
- the sieving interval (M).

If the number of primes in the factor base is small, then we do not have to find very many factorizations of $f(x)$, but they will be *very* special, there will be just few of them and it will be difficult and time consuming to find them. If we choose B large, then it will be easier to find completely factorised $f(x)$'s, but then we will need more of them. In addition, the processing of the exponents binary vectors to find linear dependencies will get more complex. So there is an optimal value for the length of the factor base which has

been proved to be about

$$B = \left[\left(e^{\sqrt{\ln N \ln \ln N}} \right)^{\frac{\sqrt{2}}{4}} \right]$$

The optimal value M for the sieving interval turns out to be about the cube of the value of B, thus

$$M = B^3 = \left[\left(e^{\sqrt{\ln N \ln \ln N}} \right)^{\frac{3\sqrt{2}}{4}} \right]$$

and

$$\left[[\sqrt{N}] - M, [\sqrt{N}] + M \right] = R$$

For the sieving procedure, the initialization of the $[log_2 f(x)]$ array can be done in R steps, the locations of the x's for which $x \equiv r_1$ or $r_2 \pmod{p}$ form two arithmetic progressions with difference p, thus may be accessed with $2 + (2R/p)$ additions of p. At each location we subtract $[\log_2 p]$, so there are $4 + (4R/p)$ steps with prime p. Also we add another $4 + (4R/p)$ steps for reading and writing to the memory. Thus there is a total of

$$\sum_p (8 + 8R/p) \approx 8B + 4R \log\log B$$

steps involving low precision additions, subtractions and memory access. After sieving, we still need to scan the resulting array in the search for the *probably* factorisable $f(x)$'s, which together with the initialization add a total of $2R$ to the above estimation of low precision steps.

Another important variable will be the Threshold which will allow for some error after the sieving is done, an will tell us which $f(x)$'s are good candidates for factorization using trial division. A good estimate for the Threshold value is

$$\frac{1}{2} \ln N + \ln M - T \ln p_{max}$$

where p_{max} is the highest prime in the factor base, and T is some value around 2 depending on the length of the number to be factorised. Silverman [18] suggest $T = 1.5$ for 30-digits, $T = 2$ for 45-digits and $T = 2.6$ for 66-digits numbers.

As shown by the author in [3], the expected running time of the algorithm is about

$$exp((1 + O(1))(\log N \log\log N)^{1/2})$$

2.4.4 Example

Find the factors of $N = 87563$ using Quadratic Sieve. If we use the optimal values recommended above we get a factor base length $B = 6$, comprising -1 and the first primes with $(N/p) = 1$ so $B = \{-1, 2, 3, 13, 17, 19\}$, also we have a value for $M = 239$,

and $Threshold = 8$. Then we compute the roots of the quadratic congruence

$$N \equiv r^2 \pmod{p}$$

for each $p \in B$ and we get

p	2	3	13	17	19
r	1	1,2	5,8	7,10	5,14

Once we have the roots we can perform the sieving over our range, which will tell us which $f(x)$'s have a chance of completely factorise over the factor base, then we use trial division on these $f(x)$'s to get the binary vectors corresponding to the exponents of the factors, so we have

x	$f(x)$	-1	2	3	13	17	19
109	-75582	1	1	0	1	1	1
242	-28899	1	0	0	0	0	1
265	-17238	1	1	1	0	1	0
296	153	0	0	0	0	1	0
299	1938	0	1	1	0	1	1
316	12393	0	0	0	0	1	0
347	32946	0	1	1	0	0	1

We have a factor base of length 6 and 7 $f(x)$'s completely factorised, so we reduce the matrix of exponents vectors to get at least 1 linearly dependent vector, $\vec{V}(ex_i) = \vec{0}$, which will be the result of a combination of $f(x)$'s being a square. To keep track of the operations performed on the Gaussian elimination we augment the exponents matrix with a (7×7) identity matrix. So we have

$$ex \left(\begin{array}{cccccc|ccccccc} 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) id$$

$$\Downarrow$$

$$Gauss\ elimination \pmod 2$$

$$\Downarrow$$

$$
ex \left(\begin{array}{cccccc|ccccccc}
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1
\end{array}\right) id
$$

Now we have 3 vectors linearly dependent, namely, $\vec{V}(ex_5)$, $\vec{V}(ex_6)$, $\vec{V}(ex_7)$, so we try a factorization with one of them, say $\vec{V}(ex_5)$. Pick up the adjoint vector from the result of the operations performed in the identity matrix

$$\vec{V}(ex_5) = \vec{0} \longrightarrow \vec{V}(id_5) = (0, 1, 1, 0, 1, 0, 0)$$

and it tells us that the product of the 2^{nd}, 3^{rd} and 5^{th} $f(x)$'s from the accumulated array of factorised $f(x)$'s is a square. We try

$$\begin{aligned}
Y^2 &= (-28899) \times (-17238) \times 1938 = 965435944356 \\
Y &= \sqrt{Y^2} \pmod N = 20473 \\
X &= 242 \times 265 \times 299 \pmod N = 20473
\end{aligned}$$

and we see that $X = Y$ will lead to a trivial factorization of N, therefore we pick up the next $\vec{0}$ vector from the reduced exponents matrix,

$$\vec{V}(ex_6) = \vec{0} \longrightarrow \vec{V}(id_6) = (0, 0, 0, 1, 0, 1, 0)$$

$$\begin{aligned}
Y^2 &= 153 \times 12393 = 1896129 \\
Y &= \sqrt{Y^2} \pmod N = 1377 \\
X &= 296 \times 316 \pmod N = 6073
\end{aligned}$$

now X and Y are different so we can try to apply Euclid's algorithm on $X - Y$ and N an see if it yields a non trivial factor of N

$$gcd(6073 - 1377, N) = 587$$

therefore we found a factorization of $N = 587 \cdot 149$.

2.4.5 Resume of the algorithm

There are basically three steps for the Quadratic Sieve factoring algorithm:

- Initialise the algorithm for a given N to be factored: find a factor base of primes with Legendre symbol $(N/p) = 1$; find a sieving range and an optimum Threshold values.
- Sieve: find the roots of N for each prime and use them to find x's in the sieving range for which $p \mid f(x)$; perform trial division over those $f(x)$ below some threshold until

we have more $f(x)$'s completely factorised than primes in the factor base; use the exponents of the factors of each factorised $f(x)$ to build binary vectors and construct a matrix with them.

- Reduce the matrix: use Gaussian elimination over the exponents matrix to get some linearly dependent vector, which will lead to a combination of $f(x)$'s being a square; use this square to solve Legendre's congruence (3) and get a factor of N.

2.5 Lenstra's Elliptic Curves Method (ECM)

Before introducing Lenstra's factoring algorithm [20, 2, 4, 22, 21], I'd like to explain briefly what is an elliptic curve [22] and some of its properties.

An elliptic curve, E, is a set of points (x, y) satisfying

$$y^2 = x^3 + ax + b \tag{9}$$

Together with the points in the curve, there is a special point, I, known as the *point at infinity*, which may be thought of as $(0, \infty)$, having an infinite y-value. The set of points on an elliptic curve, including I, form a group under the following operation. Let S be the set of points lying in the curve (9)

$$E = S \cup \{I\}$$

then given two points $A, B \in E$, if we draw a line crossing A and B, this line will pass through a third point, C. If further we draw a line crossing C and I, it will intersect the curve at a new point $A+B$, which is the reflexion on the x-axis of the point C. This curve form an Abelian group $(E, +)$ with identity element I. For a complete proof of group axioms see [18, 4]. For the purpose of Lenstra's algorithm, we define the negative of A and the sum $A + B$ as follows: for any points $A, B \in E$,

1. $A + I = A$, $I + I = I$
2. If A and B has the same x coordinate but $A \neq B$, then $A + B = I$
3. $A = (x_1, y_1)$, $B = (x_2, y_2)$, then we define $A + B = (x_3, y_3)$ as

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \tag{10}$$

$$y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \tag{11}$$

4. If $A = -B$, then $A + B = I$

5. If $A = B$, $A = (x_1, y_1)$, then let l be the tangent line to the curve at point A, and C the other point of intersection of l with the curve, then $2A = -C = (x_3, y_3)$ is

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \tag{12}$$

$$y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \tag{13}$$

Note that when working with the factoring algorithm, the group of the elliptic curve E and the operations on the points P are reduced $\pmod N$, thus the coefficients of E and each of the above formulaes for x_3 and y_3 are reduced $\pmod N$.

2.5.1 Factoring with elliptic curves

Suppose we have an elliptic curve E of the form of $y^2 = x^3 + ax + b$ with a, $b \in \mathbf{Z}$, and a point $P = (x, y)$ which satisfies it. E and P can be randomly generated by choosing three integers a, x, y in some range and then setting $b = y^2 - x^3 - ax$. Once we have a suitable E [23] we must verify that it is an elliptic curve modulo any $p \mid N$, i.e., that the cubic on the rhs has distinct roots $\pmod p$, which will hold if the discriminant $4a^3 + 27b^2$ is prime to N. Thus if $\gcd(4a^3 + 27b^2, N) = 1$, then we have a valid curve for the algorithm. If by blind of luck this *gcd* is between 1 and N, then we found a factor of N and we are done, but if it is N, then we must choose a different E.

We attempt to use E and P to factor N as explained below, but if we fail, we choose another pair (E, P) and continue until we find a factor of N. If the probability of failure is $\rho < 1$, then the probability of failure of h successive choices of (E, P) is ρ^h, which becomes very small for h large. Thus with high probability this algorithm will factor N in a reasonable number of trials.

Once we have E and P we choose an integer k which is divisible by powers of small primes below some limit (B_p), which are less than some bound (B_u). That is

$$k = \prod_{b \le B_p} b^{\alpha b}$$

where $\alpha b = [\log B_u / \log b]$ is the largest exponent such that $b^{\alpha b} \le B_u$. Then we attempt to compute $kP \pmod N$ with (12), (13) and the repeated doubling method ($2P$, $2(2P)$, ..., $2^{\alpha_2}P$, $3(2^{\alpha_2})P$, ...). If, while trying to find the inverse of $2y_1$ in (12) using Euclid's algorithm in any of the partial sums $k_sP \pmod N$ of $kP \pmod N$ we find a number which is not prime to N, that *gcd* will be a proper divisor of N, unless it happens to be N itself. Thus as soon as we try to compute $k_sP \pmod N$ for a k_s with $k_sP \pmod p = I \pmod p$ for some $p \mid N$, we will obtain a factor of N. In this case k_s is a multiple of the order of $P \pmod p$.

2.5.2 Choosing values

B_u is a bound for the product of powers of prime divisors $p \mid N$ for which we expect to find a relation $kP \pmod p = I \pmod p$, i.e., a bound for k. Using *Hasse's Theorem*[2], this bound is likely to be $p+1+2\sqrt{p} < B_u$ where $p \approx \sqrt{N}$, the bound of F_p-points on an elliptic curve. Another approach to this bound is to compute $B_u = LCM(1, ..., K)$, where $K = (\log N * \log N)/2$, and put $k = B_u$. B_p is a bound to the small primes dividing k, thus if B_p is large, there is a high probability that a pair (E, P) will accomplish the above relation, but it will take longer to compute $kP \pmod p$, so B_p has to be chosen in order to minimise the running time. k is derived from B_p and B_u.

2.5.3 Example

Factorise $N = 10403$ using Lenstra's algorithm. First we choose suitable bounds B_p and B_u. For practical purposes, we use 2 as the only prime to which we multiply the point P, if we use the second approach to find B_u, we get

$$K = \left[\frac{\log N \cdot \log N}{2}\right] = 8$$

$$k = B_u = LCM(1, 2, 3, 4, 5, 6, 7, 8) = 840$$

so we have $k = 840$.

We use the point $P = (1, 1)$ as the initial point, and try to compute a valid elliptic curve (9) with initial values: $x = 1$, $y = 1$, $a = 1$, $b = y^2 - x^3 - ax$, i.e., $b = -1$. The discriminant $4a^3 + 27b^2 = 31$ and $\gcd(31, N) = 1$ verify that this is a valid curve.

Now we try to compute $kP \pmod N$, i.e., $840(1, 1) \pmod N$ and if at any step of the process we get a $gcd \neq 1$ while trying to compute the inverse that we need in (10) or (12), this gcd will be either a non trivial factor of N or N itself, in the later case we'll have to start again with a different k and a new curve.

We write k as the sum of powers of 2, so $k = 2^3 + 2^6 + 2^8 + 2^9$ and

$$kP \pmod N = 2^3P + 2^6P + 2^8P + 2^9P \pmod N$$

We start the iteration, using (12) and (13) we get

$$\begin{aligned} 2^3(1,1) &= (1812, 5129) \pmod N \\ 2^6(1,1) &= (1090, 7976) \pmod N \end{aligned}$$

[2]See [4], p.158

now we can add them both with (10) and (11) to obtain

$$2^3(1,1) + 2^6(1,1) = (9282, 495) \pmod{N}$$

As no gcd between 1 and N has become yet, we continue

$$2^8(1,1) = (1329, 7195) \pmod{N}$$

$$2^3(1,1) + 2^6(1,1) + 2^8(1,1) = (6134, 4080) \pmod{N}$$

$$2^9(1,1) = (3306, 6121)$$

at this step, when we try to compute $(6134, 4080) + (3306, 6121)$ we find

$$x_2 - x_1 = 3306 - 6134 = 7575 \pmod{N}$$

and $\gcd(7575, N) = 101$ which is a factor of N, thus $N = 101 \cdot 103$.

2.5.4 Running Time

Lenstra [20] shows that the expected time of the algorithm is about

$$exp((1 + o(1))\sqrt{\log N \log\log N}) \text{ (where } o(1) \to 0 \text{ as } n \to \infty)$$

in the worst case, i.e., when N is the product of two primes of the same order of magnitude. However, this algorithm has the advantage that is performs much faster than any other when N has some small prime factor p, arriving to an expected running time of order

$$exp((2 + o(1))\sqrt{\log p \log\log p})$$

Chapter 3

Play Time

I'm stuck in this dream
it is changing me
I am becoming

3.1 Real

So far we have seen very useful techniques to split a big number into the product of smaller ones, but the running time of this algorithms becomes very long for numbers over 100 digits. Are this numbers very big?, why can't we factorise them in polynomial time?. Being the multiplication a very simple algorithm known by us all, why there isn't similar methods for factoring numbers?.

It seems that a 100 digits number is not big enough, there are infinitely many bigger ones, thousand digits, trillion digits, we have no limit in the length of the numbers we can use for any purpose, and if a limit exist, it is the one imposed by the machines we use to work with such numbers. A 100 digits number fits very well in the memory of any home computer nowadays. This relatively big numbers are heavily used in securing electronic communications along public computer networks such as the Internet, like credit card transactions, public key encryption, etc, they move along network wires at the speed of light to achieve the success of some authentication and communications electronic protocols.

Actual factorization algorithms are stuck trying to minimise its running time by means of adding complicated procedures to the ones shown here. Silverman [18] describes the Multiple Polynomial Quadratic Sieve which by switching between different polynomials carefully chosen instead of using only one as in Pomerance's QS, slightly speed up the algorithm, arriving to an expected running time of order

$$exp\left((1+o(1))(\log N)^{1/2}(\log\log N)^{1/2}\right)$$

Lenstra & Lenstra [19, 13] describe the Number Field Sieve, known as the faster factorization algorithm for composite numbers with big prime factors, with an expected running time of order

$$exp\left((c+o(1))(\log N)^{1/3}(\log\log N)^{2/3}\right)$$

Both of them, together with Lenstra's ECM from Section 2.5 are actually studied and modified by many mathematicians and computer programmers around the world in a race to speed up the problem of factorization, but, although these algorithms are very powerfull tools to work out factorizations of large integers, they are still working with integer numbers, Legendre's congruence as the goal to find, quadratic residues, factor bases, greatest common divisors, factorization inside factorization, linear dependencies, etc., which I clasify as heavy weight information.

Is is clear that when multiplying two integer numbers we don't use reals at any step of the process, but the actual state of factorization algorithms is so complicated that they are no longer searching for a simple solution to this problem, instead they try to gather relations between big amounts of numbers, relying in powerful machines with big storage capabilities and fast and expensive processors to achieve the goal. Maybe it is time to start from scratch, the running time still grows exponentially.

A simple overview to this paper may have call the attention over an obvious fact about some shared property between all the algorithms, they all work in the domain of integers, **Z**. When other type of number appears in the way, it is automatically truncated to integer. But, what happen with all the information at the right hand side of the point in the result of a square root operation?, nobody cares about it, it is simply thrown out of the loop, no questions. But why?, it is information equally good than that to the left of the point, often greater, both parts together form the special number which is the square root of any other number, e.g.,

$$\sqrt{50410577699919163} = 224523000.380627291758801692406...$$

3.2 Tanto

Although integer numbers must have very rich properties that are already unknown to me, we can see clearly that the set of integer numbers is sub contained in a bigger one, the complex, although I'd like to concentrate now in the sets of rational and real numbers.

$$\mathbf{Z} \subset \mathbf{Q} \subset \mathbf{R} \subset \mathbf{C}$$

Maybe we only use integers in factorization because the input number and output factors are in integer format. But it does not mean that we should only use integers in the search for new paths to the solution of factorizations. In fact, there are more than one real number involved in the game of factorization, the first obvious one is the square root of

K, the number to be factorised. Assuming K is composite of only two big prime integers, $p_1 < p_2$, its square root, $\sqrt{K}$, will lie between the two primes,

$$p_1 < \sqrt{K} < p_2$$

with,

$$\sqrt{K} - p_1 \ < \ p_2 - \sqrt{K}$$

Once we are in $\sqrt{K}$ we are at certain distance of both primes, to the left we find p_1, to the right p_2. To move an approximate distance to the left in the search of p_1 implies to look for a scale factor that multiplied to $\sqrt{K}$ will take us just at the point of any real number with integer part $= p_1$. This introduces a second real number, $Tanto$, which we may think as the real scale factor that we need to move the $\sqrt{K}$ to the left to give us the smaller prime p_1. This number lies between 0 and 1, we denote it T, its unique real value is

$$T = \frac{\sqrt{K}}{p_2} \ , \ T_1 \in (0,1) \subset \mathbf{R} \tag{14}$$

Now we have

$$p_1 = T \times \sqrt{K} \ , \ \frac{\sqrt{K}}{T} = p_2$$

Together with T_1 we find the *inverse of Tanto*, T^{-1}, the real scale factor which can move $\sqrt{N}$ to the right towards p_2

$$p_1 = \frac{\sqrt{K}}{T^{-1}} \ , \ \sqrt{K} \times T^{-1} = p_2$$

We now have Tanto, which is dependent on the primes composing the number to be factorised and therefore unknown as the primes themselves. But we know it is there, between 0 and 1, and we know is it unique for each number to be factorised, so we may think we can search for this number to achieve a factorization instead of looking for the proper integer factors.

Tanto is another factor, a real scale factor, and as soon as we find an approximation to this number we will get the integer factors. An approximation because being real implies that Tanto has infinitely many digits after the point, but a finite amount of this digits will be enough to get any real containing p_1, the integer prime factor we are searching for. Knowing that we need numbers between 0 and 1 with arbitrary length, I want to introduce the set $\mathbf{R}_A$ which I call the "set of real numbers with arbitrary fixed length". When we truncate a real number to a fixed amount of digits after the point we are automatically switching from the set of reals to the set of rationals, therefore $\mathbf{R}_A$ lives in the domain of rational numbers. I think in rational numbers as two integer numbers separted by a point instead of the fraction notation, i.e., $n \in \mathbf{Q}$, $n = a.b$, $a, b \in \mathbf{Z}$, e.g., $n = 1/3 = 0.3'$, where "" is used to denote that 3 is periodic after the point. Thus now we are ready to describe

$\mathbf{R}_A$ as

$$\mathbf{R}_A = \{n \in \mathbf{Q} \ \mid \ n = a.b, \mid b \mid = A\} \subset \mathbf{Q}$$

We do not restrict the length of a, the left side of n, as we will not be using a's largers than K at any time.

From (14) we know T has a unique real value depending on K and p_2. If we assume we don't know p_1 and p_2, but $\mid p_1 \mid = \mid [\sqrt{K}] \mid = \mid p_2 \mid$, and that we are working with numbers in $\mathbf{R}_A$, we may think in Tanto as a finite set containing all rational numbers between 0 and 1 of length A that multiplied to $\sqrt{K}$ will produce $\mathbf{R}_A$ numbers with integer part equal to p_1, thus

$$Tanto = (a, b) \subset \left(\frac{n}{\sqrt{K}}, \frac{m}{\sqrt{K}} \right) \subset (0, 1) \subset \mathbf{R}_A \tag{15}$$

$$n, m \in \mathbf{Z}^+ , \ \begin{cases} n & = & \text{min prime with} & \mid n \mid & = & \mid [\sqrt{N}] \mid \\ m & = & \text{max prime with} & m & < & [\sqrt{K}] \end{cases}$$

such that

$$T \times \sqrt{K} = p \, , \, [p]_A \in (p_1, p_1 + 1) \subset \mathbf{R}_A, \text{ for all } T \in Tanto$$

3.3 Root

There is more than Tanto. We have two real r^{th} roots which lead to the integer prime factors of K, say $r_1 \, , \ r_2 \in \mathbf{R}$ such that

$$\begin{array}{rcll} p_1 & = & \sqrt[r_1]{K} \, , & r_1 \in (2, p_1) \quad \subset \mathbf{R} \\ p_2 & = & \sqrt[r_2]{K} \, , & r_2 \in (1, 2) \quad \subset \mathbf{R} \end{array}$$

Note that the upper bound p_1 for r_1 is a very high estimate that can probably be reduced considerably. This two roots are also dependent on p_1 and p_2 such that

$$r_1 = \frac{\log K}{\log p_1} \, , \quad r_2 = \frac{\log K}{\log p_2}$$

and therefore unknown, but they have some nice properties that called my attention. If we where in the domain of positive integers, $\mathbf{Z}^+$, there are not two numbers a and b, $a \neq b$, with $a + b = a \times b$. But in the domain of positive reals there are infinitely many pairs of a's and b's satisfying this property, and r_1, r_2 are two such numbers

$$r_1 + r_2 = r_1 \times r_2$$

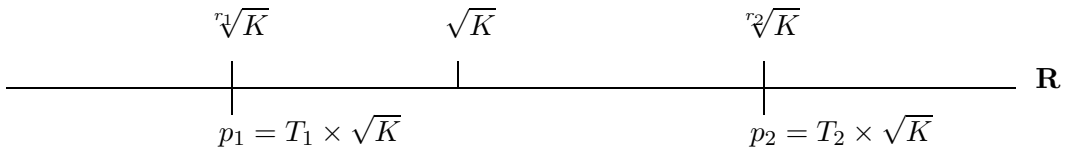
with the following relation

$$r_1 = \frac{r_2}{r_2 - 1} \, , \quad r_2 = \frac{r_1}{r_1 - 1} \tag{16}$$

Thus

$$r_1 - \frac{r_1}{r_2} = r_2 - \frac{r_2}{r_1} = 1\,,\ r_1 + r_2 - \left(\frac{r_1^2 + r_2^2}{r_1 r_2}\right) = 2$$

By now we have three real numbers directly involved in factorization, $\sqrt{K}$, T and r_1, with T^{-1} and r_2 derived from them, i.e.,



Playing a little with these numbers we find some relations such as

$$\frac{\sqrt[r_1]{T}}{\sqrt[r_2]{T^{-1}}} = \frac{\sqrt[r_2]{T}}{\sqrt[r_1]{T^{-1}}} = T \implies (1)\ \sqrt[r_1]{T} = T\,\sqrt[r_2]{T^{-1}}$$
$$\implies (2)\ \sqrt[r_2]{T} = T\,\sqrt[r_1]{T^{-1}}$$

$$\begin{aligned}(1) &\implies r_1 = \frac{\log T}{\log\left(T\,\sqrt[r_2]{T^{-1}}\right)} = \frac{\log T}{\log\left(T/T^{(r_1-1)/r_1}\right)}\\ &\implies r_1 \times \log T - (r_1 - 1) \times \log T = \log T\\ (2) &\implies r_2 \times \log T - (r_2 - 1) \times \log T = \log T\end{aligned}$$

thus

$$r_1 \times \log T - (r_1 - 1) \times \log T = r_2 \times \log T - (r_2 - 1) \times \log T$$

Now we have r_1 and r_2 in terms of each other and T, if we use (16) we have

$$\log T = \frac{\log T}{r_1} + \frac{\log T}{r_2}$$

$$\frac{\log T}{r_2} = \log T - \frac{\log T}{r_1}$$

$$r_2 = \frac{\log T}{\log T - \dfrac{\log T}{r_1}} = \frac{r_1}{r_1 - 1} = \frac{\log K}{\log p_2}$$

and equally

$$r_1 = \frac{\log T}{\log T - \dfrac{\log T}{r_2}} = \frac{r_2}{r_2 - 1} = \frac{\log K}{\log p_1}$$

If we assume again that we are working with the set $\mathbf{R}_A$ in the search for r_2, which we know lies between 1 and 2, we may think that there exists a set

$$(c, d) \subset (1, 2) \subset \mathbf{R}_A \tag{17}$$

containing all approximations to r_2 with arbitrary length A, such that

$$\text{for all } r \in (c,d)\,,\ \sqrt[r]{K} = p\,,\ [p]_A \in (p_2, p_2+1) \subset \mathbf{R}_A$$

We now use (15,17) to extend the Tanto set so it will will contain both approximations to T and to r_2, which proved to be important numbers in factorization, thus

$$Tanto = (a,b) \cup (c,d)$$

It is not clear why is this information useful, or if it useful at all, but relations between these real numbers involved in factorization keep comming the more we try, so maybe we can continue gathering relations and information while trying to devise a computer algorithm capable of performing arithmetic with the set of numbers in $\mathbf{R}_A$ in the search for any element included in Tanto.

Techniques such as Genetic Algorithms [26] may be helpful to build a random algorithm which evolves from one generation to another in a search for any of this Tanto approximations.

3.4 Complex

Briefily mention here the announcement by Peter Shor [24, 25] of a polynomial time factoring algorithm using techniques from quantum computing [27], exploiting the vastness of its powerfull parallel capabilities while working in the field of complex numbers. No quantum computer has been already built to explore this algorithm, but the theory of the algorithm is complete and waiting for one of this machines where to try an implementation.

Quantum computing is a fascinating field full of gaps which we may fill with new algorithms where to exploit the richness of complex numbers and quantum information.

Recall my interest to learn in depth the properties of complex numbers, driven by the attraction of its imaginary part.

Appendix A

Useful Background

in muddle love
me feel so strong
now you know

A.1 Quadratic Residues

Given integers N and a with $(N, a) = 1$, if the congruence

$$x^2 \equiv a \pmod{N}$$

has solutions x, then a is called a *quadratic residue* of N. If the congruence has no solutions, then a is a *quadratic non residue* $\pmod{N}$. When N is an odd prime number, Legendre introduced the symbol (a/N) which is given the value $+1$ if a is a quadratic residue, and -1 if a is a quadratic non residue of N.

Example: Find all quadratic residues $\pmod{7}$
Square each integer < 7 and look to the residues:

$$1^2 \equiv 1 \pmod{7}, \; 2^2 \equiv 4 \pmod{7}, \; 3^2 \equiv 2 \pmod{7}$$

$$4^2 \equiv 2 \pmod{7}, \; 5^2 \equiv 4 \pmod{7}, \; 6^2 \equiv 1 \pmod{7}$$

Thus 1, 2, 4 are quadratic residues $\pmod{7}$ and 3, 5, 6 are quadratic non residues $\pmod{7}$.

A.2 Continued Fraction Expansions

A continued fraction is an expression of the form:

$$b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cdots + \cfrac{a_n}{b_n}}}} = b_0 + \frac{a_1|}{|b_1} + \frac{a_2|}{|b_2} + \cdots + \frac{a_n|}{|b_n}$$

where the numbers a_i are *partial numerators* and the numbers b_i (apart form b_0) are *partial denominators*. When all a_i's are equal to 1, b_0 is an integer and all b_i's are positive integers, the continued fraction is said to be *regular*. The computation of the regular fraction expansion of $\sqrt{N}$

$$\sqrt{N} = b_0 + \frac{1|}{|b_1} + \frac{1\,|}{|\,b_2} + \frac{1\,|}{|\,b_3} + \frac{1\,|}{|\,b_4} + \cdots$$

can be formally found applying the following equalities and recursive relations:

$$x_i = \frac{Q_i - 1}{\sqrt{N} - P_i} = \frac{\sqrt{N} + P_i}{Q_i} = b_i + \frac{\sqrt{N} - P_{i+1}}{Q_i} \tag{18}$$

where

$$b_i = [x_i] = \left[\frac{\sqrt{N} + P_i}{Q_i}\right],\ P_{i+1} = b_i Q_i - P_i \tag{19}$$

If N is a rational number then the expansion will terminate with one x_i being exactly an integer, otherwise the expansion is infinite. The fraction expansion is always periodic [1].

Example: $N = 69$, $x_0 = \sqrt{69}$, $b_0 = \left[\sqrt{69}\right] = 8$

$$\begin{aligned}
x_1 &= \frac{1}{\sqrt{N} - 8} = \frac{\sqrt{N} + 8}{5} = 3 + \frac{\sqrt{N} - 7}{5} \\
x_2 &= \frac{5}{\sqrt{N} - 7} = \frac{\sqrt{N} + 7}{4} = 3 + \frac{\sqrt{N} - 5}{4} \\
x_3 &= \frac{4}{\sqrt{N} - 5} = \frac{\sqrt{N} + 5}{11} = 1 + \frac{\sqrt{N} - 6}{11} \\
&\vdots \\
x_7 &= \frac{4}{\sqrt{N} - 5} = \frac{\sqrt{N} + 7}{5} = 3 + \frac{\sqrt{N} - 8}{5} \\
x_8 &= \frac{5}{\sqrt{N} - 8} = \frac{\sqrt{N} + 8}{1} = 16 + \frac{\sqrt{N} - 8}{1} \\
x_9 &= \frac{1}{\sqrt{N} - 8} = \frac{\sqrt{N} + 8}{5} = 3 + \frac{\sqrt{N} + 7}{5}
\end{aligned}$$

Here the partial denominators b_i in the expansion are the initial integers of the right most expressions of each line. Since $x_9 = x_1$, the previous calculation will be repeated from this point onwards. So the entire expression is periodic and we express it as:

$$\sqrt{N} = 8 + \frac{1\,|}{|\,3} + \frac{1\,|}{|\,3} + \frac{1\,|}{|\,1} + \frac{1\,|}{|\,4} + \frac{1\,|}{|\,1} + \frac{1\,|}{|\,3} + \frac{1\,|}{|\,3} + \frac{1\,|}{|\,3} + \frac{1\,|}{|\,16} + \frac{1\,|}{|\,3}$$

A.3 Shanks-Tonelli algorithm

This algorithm is used to find the two roots of a quadratic congruence of the form

$$N \equiv r^2 \pmod{p} \tag{20}$$

when Legendre symbol $(N/p) = 1$, in other words, when p is a quadratic residue $\pmod{N}$.

1. BEGIN with an integer a and a prime $p > 2$, relatively prime to a. Calculate $a^{(p-1)/2} \pmod{p}$. Now $a^{(p-1)/2} \equiv 1 \text{ or } -1 \pmod{p}$.
2. IF $a^{(p-1)/2} \equiv -1 \pmod{p}$, then a has no square root $\pmod{p}$. Choose another a and go to 1.
3. IF $a^{(p-1)/2} \equiv \pmod{p}$, write $p - 1 = s \cdot 2^e$ with s odd and e positive.
4. FIND a number n such that $n^{(p-1)/2} \equiv 1 \pmod{p}$, that is, a non square $\pmod{p}$.
5. INITIALIZE these variables $\pmod{p}$

$$\begin{aligned} x &\equiv a^{(p-1)/2} \\ b &\equiv a^s \\ g &\equiv n^s \\ r &= e \end{aligned}$$

6. WHILE $m > 0$
   - FIND the least integer m such that $0 \leq m \leq r - 1$ and $b^{2^m} \equiv 1 \pmod{p}$.
   - IF $m = 0$ then RETURN the value of x and EXIT.
   - IF $m > 0$ then REPLACE the variables:

$$\begin{aligned} x &\leftarrow x \cdot g^{2^{r-m-1}} \\ b &\leftarrow b \cdot g^{2^{r-m}} \\ g &\leftarrow g^{2r-m} \\ r &\leftarrow m \end{aligned}$$

Note that the algorithm returns only one of the roots of the congruence (20), say r_1, the other is $r_2 = p - r_1$.

Appendix B

Zero Day

tried so hard to make the pieces don't fit
where are you?
you and me
make it through
where the hell are you?

B.1 Adults Game

In this chapter I present C++ source code implementing some of the factorization algorithms previously described.

For the purpose of this project, I've been supported by an AMD 500 Mhz processor, 256 MB RAM, 6 GB HD Hardware, and a GNU/Linux Redhat 8.0 Operating System including the GMP library, C++ development libraries and compiler.

At the beginning I faced the limit of computer arithmetic in my 32 bits processor, but some research pointed me in the direction of the GMP library (GNU Multiple Precision). Of course, many people before me had similar ideas thus many of the tasks I had to do to implement the GMP library into my programs was already done and well documented. I found a particularly useful open source class called Integer. This class is a complete C++ wrapper for the GMP library, with all operators overloaded and many useful functions linked to number theory. I studied, structured and modified it to suit my needs.

All source code printed here can be found in the CD-ROM attached to this document. Anyone willing to use it will need a C++ compiler and the GMP library, which are freely distributed with any GNU/Linux distribution. The Integer C++ class (Integer.h, Integer.cc) is included with the source code. Screenshots on each section will tell you how to compile and run the programs. The algorithms arrive to consume 99% of processor time, but just few Mb of memory. To factorise a number greater than 70 digits long with only two large prime factors, prepare to wait for an output from some hours to few days or weeks.

B.2 Trial Division Algorithm

```
/*************************************************************
 *  Trial_division.cc                                        *
 *    Factor any integer using trial division                *
 *    algorithm with 2, 3, 6k-+1 for k=1,2,..                *
 *                                                           *
 *  Compile:                                                 *
 *    g++ -g -o Trial Trial_division.cc Integer.o -lgmp      *
 *                                                           *
 *  Run:                                                     *
 *    ./Trial <number_to_factor>                             *
 *************************************************************/

#include <sys/time.h>
#include "Integer.h"

Integer get_factor( Integer ) ;

int main(int argc, char *argv[]) {

    // Structures to control initial,
    // final and total running time
    struct timeval * tv1 = new timeval ;
    struct timeval * tv2 = new timeval ;
    struct timeval * tvt = new timeval ;
    struct timezone * tz = NULL ;

    Integer key , factor ;

    // get start time
    gettimeofday(tv1,tz);

    cout << "\n\tTrial Division Factoring Algorithm\n" ;
    cout << "\t---------------------------------\n" << flush ;

    if ( argc < 2 ) {
        cerr << "Usage: " << argv[0] << " <number_to_factor>" << endl ;
        return -1 ;
    }

    key = argv[ 1 ] ;
```

```
    factor = 5 ;
    cout << "Number to factor: " << key << endl ;
    cout << "Factors found: " << flush ;

    if ( isprime( key ) ) {
        cout << key << " is prime !" << endl ;
        return 0 ;
    }

    // main loop
    while ( ! isprime( key ) ) {
        factor = get_factor( key ) ;
        cout << factor << " x " << flush ;
        key /= factor ;
    }
    cout << key << endl ;

    // get finish time
    gettimeofday(tv2,tz);
    // get total time
    timersub(tv2,tv1,tvt);
    cout << "Total Running time: " << tvt->tv_sec << "s " ;
    cout << tvt->tv_usec << "mcs\n\n" ;

    return 0 ;
}

Integer get_factor( Integer key ) {

    Integer factor = 5 ;
    if ( key % 2 == 0 ) return 2 ;
    if ( key % 3 == 0 ) return 3 ;

    for ( ; ; ) {
        if ( key % factor == 0 ) {
            if ( isprime( factor ) ) return factor ;
            else return get_factor( factor ) ;
        }
        else if ( key % ( factor + 2 ) == 0 ) {
            if ( isprime( factor + 2 ) ) return ( factor + 2 ) ;
            else return get_factor( factor + 2 ) ;
        }
```

```
        else {
            factor += 6 ;
        }
    }
}

/* <rou@papelmail.com> */
```

B.2.1 Screenshot

```
[r@localhost code]$ g++ -ggdb -c Integer.cc
[r@localhost code]$ g++ -ggdb -o Trial Trial_division.cc Integer.o -lgmp
[r@localhost code]$ ./Trial 2345678

    Trial Division Factoring Algorithm
    ----------------------------------
    Number to factor: 2345678
    Factors found: 2 x 23 x 50993
    Total Running time: 0s 50704mcs

[r@localhost code]$ ./Trial 23456789991

    Trial Division Factoring Algorithm
    ----------------------------------
    Number to factor: 23456789991
    Factors found: 3 x 3 x 27449 x 94951
    Total Running time: 0s 240149mcs

[r@localhost code]$ ./Trial 23456789991897911

    Trial Division Factoring Algorithm
    ----------------------------------
    Number to factor: 23456789991897911
    Factors found: 11 x 4313653 x 494345617
    Total Running time: 10s 171169mcs

[r@localhost code]$ ./Trial 23456789991897911337

    Trial Division Factoring Algorithm
    ----------------------------------
    Number to factor: 23456789991897911337
    Factors found: 3 x 13 x 59 x 71 x 523 x 274531335889
```

```
    Total Running time: 0s 7744mcs

[r@localhost code]$ ./Trial 234567899918979113376573443

    Trial Division Factoring Algorithm
    ----------------------------------
    Number to factor: 234567899918979113376573443
    Factors found: Killed

[r@localhost code]$ exit
```

B.3 Fermat's Algorithm

```
/*************************************************************
 *  Fermat.cc                                                *
 *    Uses Fermat method to factor a large Integer           *
 *                                                           *
 *  Compile:                                                 *
 *    g++ -g -o Fermat Fermat.cc Integer.o -lgmp             *
 *                                                           *
 *  Run:                                                     *
 *    ./Fermat <number_to_factor>                            *
 *************************************************************/

#include <sys/time.h>
#include "Integer.h"

Integer get_factor( Integer ) ;

int main ( int argc , char * argv[] ) {

    // Structures to control initial,
    // final and total running time
    struct timeval * tv1 = new timeval ;
    struct timeval * tv2 = new timeval ;
    struct timeval * tvt = new timeval ;
    struct timezone * tz = NULL ;

    Integer key , factor ;

    // get start time
    gettimeofday(tv1,tz);

    cout << "\n\tFermat's Factoring Algorithm\n" ;
    cout << "\t---------------------------\n" << flush ;

    if ( argc != 2 ) {
        cerr << "Usage: " << argv[0] << " <number_to_factor>" << endl ;
        return -1 ;
    }

    key = argv[ 1 ] ;
    cout << "Number to factor: " << key << endl ;
```

```
    cout << "Factors found: " << flush ;

    if ( isprime( key ) ) {
        cout << key << " is prime !" << endl ;
        return 0 ;
    }

    // main loop
    while ( ! isprime( key ) ) {
        factor = get_factor( key ) ;
        cout << factor << " x " << flush ;
        key /= factor ;
    }
    cout << key << endl ;

    // get finish time
    gettimeofday(tv2,tz);
    // get total time
    timersub(tv2,tv1,tvt);
    cout << "Total Running time: " << tvt->tv_sec << "s " ;
    cout << tvt->tv_usec << "mcs\n\n" ;

    return 0 ;
}

Integer get_factor( Integer key ) {

    Integer root, diff , trpone ;

    if ( key % 2 == 0 ) return 2 ;
    if ( key % 3 == 0 ) return 3 ;
    if ( key % 5 == 0 ) return 5 ;

    root = Sqrt( key ) ;
    if ( issquare( key ) ) {
        if ( isprime( root ) ) return root ;
        else return get_factor( root ) ;
    }

    root ++ ;
    trpone = ( 2 * root ) + 1 ;
    diff = ( root * root ) - key ;
```

```
    while ( ! issquare( diff ) ) {
        diff += trpone ;
        trpone += 2 ;
        root ++ ;
    }
    return ( root + Sqrt( diff ) ) ;
}

/* <rou@papelmail.com> */
```

B.3.1 Screenshot

```
[r@localhost code]$ g++ -ggdb -c Integer.cc
[r@localhost code]$ g++ -ggdb -o Fermat Fermat.cc Integer.o -lgmp
[r@localhost code]$ ./Fermat 10403

    Fermat's Factoring Algorithm
    ----------------------------
    Number to factor: 10403
    Factors found: 103 x 101
    Total Running time: 0s 122998mcs

[r@localhost code]$ ./Fermat 1040301

    Fermat's Factoring Algorithm
    ----------------------------
    Number to factor: 1040301
    Factors found: 3 x 3 x 115589
    Total Running time: 0s 251992mcs

[r@localhost code]$ ./Fermat 104030103

    Fermat's Factoring Algorithm
    ----------------------------
    Number to factor: 104030103
    Factors found: 3 x 336667 x 103
    Total Running time: 4s 332104mcs

[r@localhost code]$ ./Fermat 10403010304

    Fermat's Factoring Algorithm
    ----------------------------
```

```
    Number to factor: 10403010304
    Factors found: 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 x 67841 x 599
    Total Running time: 0s 954518mcs

[r@localhost code]$ ./Fermat 1040301030403

    Fermat's Factoring Algorithm
    ----------------------------
    Number to factor: 1040301030403
    Factors found: 21707761 x 2819 x 17
    Total Running time: 256s 97017mcs

[r@localhost code]$ exit
```

B.4 Shank's Algorithm

```
/*************************************************************
 *  Shanks.cc                                                *
 *    Factor any integer using Shank's algorithm             *
 *                                                           *
 *  Compile:                                                 *
 *    g++ -g -o Shanks Shanks.cc Integer.o -lgmp             *
 *                                                           *
 *  Run:                                                     *
 *    ./Shanks <number_to_factor>                            *
 *************************************************************/

#include <sys/time.h>
#include "Integer.h"

Integer get_factor( Integer ) ;

int main ( int argc , char * argv[] ) {

    // Structures to control initial,
    // final and total running time
    struct timeval * tv1 = new timeval ;
    struct timeval * tv2 = new timeval ;
    struct timeval * tvt = new timeval ;
    struct timezone * tz = NULL ;

    Integer key , factor ;

    // get start time
    gettimeofday(tv1,tz);

    cout << "\n\tShank's SQUFOF Factoring Algorithm\n" ;
    cout << "\t---------------------------------\n" << flush ;

    if ( argc != 2 ) {
        cerr << "Usage: " << argv[0] << " <number_to_factor>" << endl ;
        return -1 ;
    }

    key = argv[1] ;
```

```
    cout << "Number to factor: " << key << endl ;
    cout << "Factors found: " << flush ;

    if ( isprime( key ) ) {
        cout << key << " is prime !" << endl ;
        return 0 ;
    }

    // main loop
    while ( ! isprime( key ) ) {
        factor = get_factor( key ) ;
        cout << factor << " x " << flush ;
        key /= factor ;
    }
    cout << key << endl ;

    // get finish time
    gettimeofday(tv2,tz);
    // get total time
    timersub(tv2,tv1,tvt);
    cout << "Total Running time: " << tvt->tv_sec << "s " ;
    cout << tvt->tv_usec << "mcs\n\n" ;

    return 0 ;
}


Integer get_factor( Integer key ) {

    Integer factor, num, sqrt, den;
    Integer Q_i, x_i, b_i, A_0, A_1, A_i ;
    int i = 0;

    if ( key % 2 == 0 ) return 2 ;
    if ( key % 3 == 0 ) return 3 ;
    if ( key % 5 == 0 ) return 5 ;

    sqrt = Sqrt( key ) ;
    if ( issquare( key ) ) return sqrt ;

    b_i = sqrt ;
    x_i = sqrt ;
```

```
    A_0 = sqrt ;
    A_1 = 1 ;
    num = 1 ;

    for( ; ; ) {
        i ++ ;
        Q_i = key - ( x_i * x_i ) ;
        Q_i /= num ;
        b_i = ( sqrt + x_i ) / Q_i ;
        x_i = ( Q_i * b_i ) - x_i ;
        num = Q_i ;
        if( issquare( Q_i ) && i % 2 == 0 ) {
            den = Sqrt( Q_i ) ;
            factor = gcd( A_i - den , key ) ;
            if( factor > 1 && factor < key ) {
                if( isprime( factor ) ) return factor ;
                else return get_factor( factor ) ;
            }
        }
        A_i = ( ( b_i * A_0 ) + A_1 ) % key ;
        A_1 = A_0 ;
        A_0 = A_i ;
    }

}

/* <rou@papelmail.com> */
```

B.4.1 Screenshot

```
[r@localhost code]$ g++ -ggdb -c Integer.cc
[r@localhost code]$ g++ -ggdb -o Shanks Shanks.cc Integer.o -lgmp
[r@localhost code]$ ./Shanks 9832187312

    Shank's SQUFOF Factoring Algorithm
    ----------------------------------
    Number to factor: 9832187312
    Factors found: 2 x 2 x 2 x 2 x 771031 x 797
    Total Running time: 0s 96947mcs

[r@localhost code]$ ./Shanks 983218731293821
```

```
    Shank's SQUFOF Factoring Algorithm
    ----------------------------------
    Number to factor: 983218731293821
    Factors found: 89383521026711 x 11
    Total Running time: 1s 497016mcs

[r@localhost code]$ ./Shanks 98321873129382123

    Shank's SQUFOF Factoring Algorithm
    ----------------------------------
    Number to factor: 98321873129382123
    Factors found: 821 x 19 x 13 x 371179 x 145139 x 3 x 3
    Total Running time: 0s 541745mcs

[r@localhost code]$ ./Shanks 9832187312938212345

    Shank's SQUFOF Factoring Algorithm
    ----------------------------------
    Number to factor: 9832187312938212345
    Factors found: 8092335236986183 x 3 x 5 x 9 x 3 x 3
    Total Running time: 1s 848933mcs

[r@localhost code]$ ./Shanks 983218731293821234521

    Shank's SQUFOF Factoring Algorithm
    ----------------------------------
    Number to factor: 983218731293821234521
    Factors found: 327739577097940411507 x 3
    Total Running time: 88s 943571mcs

[r@localhost code]$ exit
```

B.5 Control

I control you
I control you
I am the voice inside your head
I am the lie that you believe
I am the truth from which you run
I control you
I control you
I'll take you where you want to go
I'll give you all you need to know[1]

[1] *T. Reznor, NIN, Further Down The Spiral*

B.6 Pollard's ρ Algorithm

```
/*************************************************************
 *  Pollard_rho.cc                                           *
 *    Uses Pollard rho method to factorize a large Integer   *
 *                                                           *
 *  Compile:                                                 *
 *    g++ -g -o Pollard_rho Pollard_rho.cc Integer.o -lgmp   *
 *                                                           *
 *  Run:                                                     *
 *    ./Pollard_rho <number_to_factor>                       *
 *************************************************************/

#include <sys/time.h>
#include "Integer.h"

Integer get_factor( Integer, int, Integer ) ;

int main ( int argc, char *argv[] ) {

    // Structures to control initial,
    // final and total running time
    struct timeval * tv1 = new timeval ;
    struct timeval * tv2 = new timeval ;
    struct timeval * tvt = new timeval ;
    struct timezone * tz = NULL ;

    Integer key , factor , x0 ;
    int a ;

    // get start time
    gettimeofday(tv1,tz);

    cout << "\n\tPollard rho Factoring Algorithm\n" ;
    cout << "\t-------------------------------\n" << flush ;

    if ( argc != 2 ) {
        cerr << "Usage: " << argv[ 0 ] << " <number_to_factor>" << endl ;
        return -1 ;
    }

    key = argv[ 1 ] ;
```

```
    x0 = 2 ;
    a = -1 ;

    cout << "Number to factor: " << key << endl ;
    cout << "Factors found: " << flush ;

    if ( isprime( key ) ) {
        cout << key << " is prime !" << endl ;
        return 0 ;
    }

    while ( !isprime( key ) ) {
        factor = get_factor( key , a , x0 ) ;
        cout << factor << " x " << flush ;
        key /= factor ;
    }
    cout << key << endl ;

    // get finish time
    gettimeofday(tv2,tz);
    // get total time
    timersub(tv2,tv1,tvt);
    cout << "Total Running time: " << tvt->tv_sec << "s " ;
    cout << tvt->tv_usec << "mcs\n\n" ;

    return 0 ;
}


Integer get_factor( Integer key , int a , Integer x ) {

    Integer y , q , g ;

    if ( key % 2 == 0 ) return 2 ;
    if ( key % 3 == 0 ) return 3 ;
    if ( key % 5 == 0 ) return 5 ;
    if ( key % 7 == 0 ) return 7 ;

    y = x ; q = 1 ;

    for ( int i = 1 ; ; i++ ) {
```

```
        x  = ( ( x * x ) + a )  %  key ;
        y  = ( ( y * y ) + a )  %  key ;
        y  = ( ( y * y ) + a )  %  key ;
        q *= ( x - y ) ;
        q  = q % key ;

        if ( ( i % 10 ) == 0 ) {
            g = gcd( key , q ) ;
            if ( g > 1 && g < key ) {
                if ( !isprime( g ) )
                    return get_factor( g , a , x ) ;
                else return g ;
            }
        }
    }
}

/* <rou@papelmail.com> */
```

B.6.1 Screenshot

```
[r@localhost code]$ g++ -ggdb -c Integer.cc
[r@localhost code]$ g++ -ggdb -o Rho Pollard_rho.cc Integer.o -lgmp
[r@localhost code]$ ./Rho 743827

    Pollard rho Factoring Algorithm
    -------------------------------
    Number to factor: 743827
    Factors found: 7 x 106261
    Total Running time: 0s 23164mcs

[r@localhost code]$ ./Rho 74382732894

    Pollard rho Factoring Algorithm
    -------------------------------
    Number to factor: 74382732894
    Factors found: 2 x 3 x 72953 x 169933
    Total Running time: 0s 17698mcs

[r@localhost code]$ ./Rho 74382732894723

    Pollard rho Factoring Algorithm
```

```
    ------------------------------
    Number to factor: 74382732894723
    Factors found: 3 x 11 x 2254022208931
    Total Running time: 0s 5814mcs

[r@localhost code]$ ./Rho 74382732894723324567

    Pollard rho Factoring Algorithm
    ------------------------------
    Number to factor: 74382732894723324567
    Factors found: 3 x 24794244298241108189
    Total Running time: 0s 24575mcs

[r@localhost code]$ ./Rho 743827328947233245671 23

    Pollard rho Factoring Algorithm
    ------------------------------
    Number to factor: 7438273289472332456712 3
    Factors found: 3 x 10211 x 2428189628659397531
    Total Running time: 0s 35061mcs

[r@localhost code]$ ./Rho 74382732894723324567123 3433

    Pollard rho Factoring Algorithm
    ------------------------------
    Number to factor: 74382732894723324567123 3433
    Factors found: 3 x 3 x 97 x 135785501 x 62748664641116221
    Total Running time: 3s 404162mcs

[r@localhost code]$ ./Rho 74382732894723324567123 43355

    Pollard rho Factoring Algorithm
    ------------------------------
    Number to factor: 74382732894723324567123 43355
    Factors found: 5 x 33488219179 x 4442322387890229149
    Total Running time: 33s 312340mcs

[r@localhost code]$ ./Rho 74382732894723324567123 433551

    Pollard rho Factoring Algorithm
    ------------------------------
    Number to factor: 74382732894723324567123 433551
```

```
    Factors found: 17 x 2819645303767 x 155177492371634809
    Total Running time: 216s 858884mcs

[r@localhost code]$ exit
```

B.7 Pollard's $p-1$ Algorithm

```
/*************************************************************
 *  Pollard_p-1.cc                                           *
 *    Uses Pollard p-1 method to factorize a large Integer    *
 *                                                            *
 *  Compile:                                                  *
 *    g++ -g -o Pollard_p-1 Pollard_p-1.cc Integer.o -lgmp    *
 *                                                            *
 *  Run:                                                      *
 *    ./Pollard_p-1 <number_to_factor>                        *
 *************************************************************/

#include <sys/time.h>
#include "Integer.h"

Integer get_factor( Integer ) ;

int main( int argc , char * argv[] ) {

    // Structures to control initial,
    // final and total running time
    struct timeval * tv1 = new timeval ;
    struct timeval * tv2 = new timeval ;
    struct timeval * tvt = new timeval ;
    struct timezone * tz = NULL ;

    Integer key , factor ;

    // get start time
    gettimeofday(tv1,tz);

    cout << "\n\tPollard p-1 Factoring Algorithm\n" ;
    cout << "\t-------------------------------\n" << flush ;

    if ( argc < 2 ) {
        cerr << "Usage: " << argv[ 0 ] << " <number_to_factor>" << endl ;
        return -1 ;
    }

    key = argv[ 1 ] ;
```

```
    cout << "Number to factor: " << key << endl ;
    cout << "Factors found: " << flush ;

    if ( isprime( key ) ) {
        cout << key << " is prime !" << endl ;
        return 0 ;
    }

    // main loop
    while ( !isprime( key ) ) {
        factor = get_factor( key ) ;
        cout << factor << " x " << flush ;
        key /= factor ;
    }
    cout << key << endl ;

    // get finish time
    gettimeofday(tv2,tz);
    // get total time
    timersub(tv2,tv1,tvt);
    cout << "Total Running time: " << tvt->tv_sec << "s " ;
    cout << tvt->tv_usec << "mcs\n\n" ;

    return 0;
}


Integer get_factor( Integer key ) {

    Integer base , g ;

    if ( key % 2 == 0 ) return 2 ;
    if ( key % 3 == 0 ) return 3 ;
    if ( key % 5 == 0 ) return 5 ;
    if ( key % 7 == 0 ) return 7 ;

    base = 2 ; g = 1 ;

    for ( Integer i = 1 ; ; i++ ) {
        base = PowModN( base , i , key ) ;
        if ( i % 10 == 0 ) {
            g = gcd( ( base - 1 ) , key ) ;
```

```
            if ( g > 1 && g < key ) {
                if ( !isprime( g ) )
                    return get_factor( g ) ;
                else return g ;
            }
        }
    }
}

/* <rou@papelmail.com> */
```

B.7.1 Screenshot

```
[r@localhost code]$ g++ -ggdb -c Integer.cc
[r@localhost code]$ g++ -ggdb -o P-1 Pollard_p-1.cc Integer.o -lgmp
[r@localhost code]$ ./P-1 971298712

    Pollard p-1 Factoring Algorithm
    -------------------------------
    Number to factor: 971298712
    Factors found: 2 x 2 x 2 x 157 x 233 x 3319
    Total Running time: 0s 100060mcs

[r@localhost code]$ ./P-1 9712987127433211

    Pollard p-1 Factoring Algorithm
    -------------------------------
    Number to factor: 9712987127433211
    Factors found: 37 x 262513165606303
    Total Running time: 0s 5504mcs

[r@localhost code]$ ./P-1 97129871274332111

    Pollard p-1 Factoring Algorithm
    -------------------------------
    Number to factor: 97129871274332111
    Factors found: 157742069 x 615751219
    Total Running time: 0s 291533mcs

[r@localhost code]$ ./P-1 971298712743321119999

    Pollard p-1 Factoring Algorithm
```

```
    ------------------------------
    Number to factor: 97129871274332111999
    Factors found: 7 x 509 x 27260699206941373
    Total Running time: 0s 80482mcs

[r@localhost code]$ ./P-1 971298712743321119998778979312398986187

    Pollard p-1 Factoring Algorithm
    ------------------------------
    Number to factor: 971298712743321119998778979312398986187
    Factors found: 7 x Terminated

[r@localhost code]$ ./P-1 9712987127433211199987789793123989861872221

    Pollard p-1 Factoring Algorithm
    ------------------------------
    Number to factor: 9712987127433211199987789793123989861872221
    Factors found: 3 x 32376623758110703999959299310413299539574 07
    Total Running time: 0s 123671mcs

[r@localhost code]$ ./P-1 97129871274332111999877897931239898618722213

    Pollard p-1 Factoring Algorithm
    ------------------------------
    Number to factor: 97129871274332111999877897931239898618722213
    Factors found: 3 x 3 x 3 x 109 x 8116201927 x 385069 x 1471254211733
                   x 7177669073629
    Total Running time: 5s 802677mcs

[r@localhost code]$ exit
```

B.8 Pomerance's Quadratic Sieve Algorithm

```
/*************************************************************
 *             Mike Roig collaboration in                    *
 *  this algorithm and the bitset.{cc,h} class.  Feb 2003    *
 *************************************************************
 *  QS.cc                                                    *
 *    Uses Pomerance's Quadratic Sieve algorithm to factorize *
 *    a large composite Integer                              *
 *                                                           *
 *  Compile:                                                 *
 *    g++ -g -o QS QS.cc bitset.cc Integer.o -lgmp           *
 *                                                           *
 *  Run:                                                     *
 *    ./QS <number_to_factor>                                *
 *************************************************************/

#include "Integer.h"
#include "bitset.h"
#include <math.h>
#include <sys/time.h>

// Silverman et. al. and Pomerance
// Thresold, Factor Base length and Sieve range
#define LN(x) ( log2(x) * log(2) )
#define T ( 0.0268849 * len + 0.783929 )
#define M(n) (pow(exp(sqrt(LN(n)*LN(LN(n)))),3*(sqrt(2)/4)))
#define THRESHOLD(n,p) ((int) (0.7*LN(n)+log(M(n))-T*log(p)))

// Use only one of the B_LEN definitions,
// for large numbers Silverman's is recommended

// Pomerance's Base Length:
#define B_LEN ( (int) (pow(exp(sqrt(LN(key)*LN(LN(key)))),(sqrt(2)/4))) )

// Silverman's Base Length:
// #define B_LEN ( (int) (2.93 * (len * len) - 164.4 * len + 2455) )
#define EXTRA 5
#define BLOCK 20000


void init ( void ) ;
```

```
void sieve ( Integer ) ;
bool divide ( Integer , bitset & ) ;
void matrix( void ) ;
void reduce ( void ) ;

int * factor_base , base_length ;
int * roots , * logs ;
int threshold , len , siev_block ;
bitset * exmatrix , * idmatrix ;
Integer key , sqrt_key , range , * fx , * xx , * pfx , * pxx ;

main( int argc , char *argv[] ) {

    // Structures to control initial,
    // final and total running time
    struct timeval * tv1 = new timeval ;
    struct timeval * tv2 = new timeval ;
    struct timeval * tvt = new timeval ;
    struct timezone * tz = NULL ;

    // get start time
    gettimeofday(tv1,tz);

    cout << "\nPomerance's Quadratic Sieve Factoring Algorithm\n" ;
    cout << "-----------------------------------------------\n\n" ;
    cout << flush ;

    if ( argc < 2 ) {
        cerr << "Usage: " << argv[0] << " <number_to_factor> " << endl ;
        return 1 ;
    }

    key = argv[ 1 ] ;
    len = strlen( argv[1] ) ;

    cout << "Entering Init phase...\n" ;
    init() ;
    cout << "...Init phase done!\n\n" << flush ;

    cout << "Building exponent matrix...\n" << flush ;
    matrix() ;
    cout << "...exponent matrix done!\n\n" ;
```

```
    cout << "Entering Reduce phase..." << flush ;
    reduce() ;

    // get finish time
    gettimeofday(tv2,tz);
    // get total time
    timersub(tv2,tv1,tvt);
    cout << "Total Running time: " << tvt->tv_sec << "s " ;
    cout << tvt->tv_usec << "mcs\n\n" ;

    return 0 ;
}

void init ( void ) {

    sqrt_key = Sqrt( key ) ;
    cout << "\tNumber to factor: " << key << '\n' ;
    cout << "\t[ sqrt( key ) ] = " << sqrt_key << '\n' ;

    base_length = B_LEN ;
    range = M( key ) ;

    if ( 2 * range < BLOCK )
        siev_block = 2 * range ;
    else siev_block = BLOCK ;

    // init factor base
    factor_base = new int [ base_length ] ;
    cout << "\tBuilding factor base..." ;
    Integer prime = 2 ;
    int i = 0 ;
    while ( i < base_length ) {
        if ( Legendre( key , prime ) == 1 ) {
            factor_base[i] = prime ;
            ++ i ;
        }
        prime = nextprime( prime ) ;
    }
    cout << "done! \n" ;

    // init Threshold
```

```
    threshold = THRESHOLD( key , factor_base[ base_length - 1 ] ) ;

    // init roots array: key = x^2 mod p
    roots = new int [ base_length ] ;
    cout << "\tlooking for roots of N mod p..." ;
    for ( int i = 0 ; i < base_length ; ++ i ) {
        roots[i] = SqrtModN ( key , factor_base[i] ) ;
    }
    cout << "done! \n" ;

    // init logs, f(x)'s, x's arrays
    logs = new int [ siev_block ] ;
    fx = new Integer[ siev_block ] ;
    xx = new Integer[ siev_block ] ;

    // init array of good fx's and x's ;-)
    pfx = new Integer [ base_length + EXTRA ] ;
    pxx = new Integer [ base_length + EXTRA ] ;

    // init 2 matrices (exponents and id)
    exmatrix = new bitset [ base_length + EXTRA ] ;
    idmatrix = new bitset [ base_length + EXTRA ] ;
    for ( int x = 0 ; x < base_length + EXTRA ; ++ x ) {
        exmatrix[ x ].set_size( base_length ) ;
        idmatrix[ x ].set_size( base_length + EXTRA ) ;
    }
}

# define binlog(x) ( log(x) / log(2) )

void sieve ( Integer n ) {

    int p , k = 0 ;
    Integer x ;

    for ( int i = 0 ; i < siev_block ; i++ ) {
        x = n + i ;
        xx[ i ] = x ;
        x = x * x - key ;
        fx[ i ] = x ;
        logs[ i ] = log2( x ) ;
    }
```

```
    // for each factor in factor base
    for ( int i = 0 ; i < base_length ; ++ i ) {
        int r = roots[ i ] ;
        x = n ;
        p = factor_base[i] ;
        // find x congruent to r1 mod p
        while( x < ( n + siev_block ) ) {
            if( x % p == r )
                break ;
            else ++ x ;
        }
        k = (int) ( x - n ) ;
        while( k < siev_block ) {
            logs[ k ] -= (int) binlog( p ) ;
            k += p ;
        }
        // find x congruent to r2 mod p
        x = n ;
        r = p - roots[i] ;
        while( x < ( n + siev_block ) ) {
            if( x % p == r )
                break;
            else ++ x ;
        }
        k = (int) ( x - n ) ;
        while( k < siev_block ) {
            logs[ k ] -= (int) binlog( p ) ;
            k += p ;
        }
    }
}

void matrix( void ) {
    int k = 0 ;
    Integer block = 0 ;
    int needed = base_length + EXTRA ;
    cout << "\tTotal f(x)'s needed: " << needed << '\n' << flush ;
    while ( k < needed ) {
        sieve( sqrt_key - ( range - block ) ) ;
        for ( int i = 0 ; i < siev_block ; ++ i ) {
            if ( logs[i] <= threshold ) {
```

```
                if ( k < needed ) {
                    if ( divide( fx[ i ] , exmatrix[ k ] ) ) {
                        pxx[ k ] = xx[ i ] ;
                        pfx[ k ] = fx[ i ] ;
                        idmatrix[ k ].flip( k ) ;
                        k++ ;
                    }
                }
                else break ;
            }
        }
        cout << "\tSieving... f(x)'s factorized: " << k << '\r' << flush ;
        block += siev_block ;
    }
    cout << '\n' << flush ;
}

bool divide ( Integer fx , bitset & row ) {
    int trial ;
    int i = 0 ;
    if ( fx < 0 ) {
        fx = -fx ;
    }
    while ( i < base_length ) {
        trial = factor_base[ i ] ;
        if ( fx % trial == 0 ) {
            row.flip(i) ;
            fx /= trial ;
        }
        else i++ ;
    }
    if ( fx <= factor_base[ base_length - 1 ] ) {
        return true ;
    }
    else {
        row.reset() ;
        return false ;
    }
}


void reduce( void ) {
```

```
bool pivot = false ;
Integer X , Y ;
bitset * expivot , * idpivot , btmp ;
for ( int x = 0 ; x < base_length ; x++ ) {
    for ( int y = x ; y < base_length + EXTRA ; y++ ) {
        if ( !pivot && exmatrix[y][x] ) {
            // swap ex and id rows
            btmp.set_size( base_length ) ;
            btmp = exmatrix[ y ] ;
            exmatrix[ y ] = exmatrix[ x ] ;
            exmatrix[ x ] = btmp ;
            btmp.set_size( base_length + EXTRA ) ;
            btmp = idmatrix[ y ] ;
            idmatrix[ y ] = idmatrix[ x ] ;
            idmatrix[ x ] = btmp ;
            //
            expivot = & exmatrix[ x ] ;
            idpivot = & idmatrix[ x ] ;
            pivot = true ;
            y ++ ;
        }
        if ( y == base_length + EXTRA ) continue ;
        if ( pivot && exmatrix[y][x] ) {
            exmatrix[ y ] += * expivot ;
            idmatrix[ y ] += * idpivot ;
        }
    }
    pivot = false ;
}

for ( int i = base_length ; i < base_length + EXTRA ; i++ ) {
    X = 1 , Y = 1 ;
    for ( int j = 0 ; j < base_length + EXTRA ; j++ ) {
        if ( idmatrix[i][j] ) {
            Y *= pfx[ j ] ;
            X *= pxx[ j ] ;
        }
    }
    if( Y < 0 ) Y = -Y ;
    X = X % key ;
    Y = Sqrt( Y ) % key ;
    X = gcd( X-Y , key ) ;
```

```
        if ( ! ( X == 1 || X == key ) ) break ;
    }
    cout << " Factors: " << X << " x " << key/X << "\n\n" ;
}

/* <rou@papelmail.com> */
/* <rtx@papelmail.com> */
```

B.8.1 Bitset class

For the linear algebra part of the QS algorithm we modified the Integer class to add bit fitling functionality to it. But we didn't obtain what we needed so we wrote our own class implementing bit manipulation, allowing us to know exactly what was going on at debug level.

```
/*********************************************************
 * bitset.h                                              *
 *      bitset object type and operations needed by      *
 *      Pomerance's Quadratic Sieve Algorithm            *
 *                                                       *
 *      Courtesy of                                      *
 *          Mike Roig <rtx@papelmail.com>                *
 *      in collaboration with                            *
 *          R. Arigita <rou@papelmail.com>               *
 *                                                       *
 *      Canterbury, UK, Feb 2003.                        *
 *********************************************************/


# ifndef MATRIX
# define MATRIX

# include <iostream>
using std :: ostream ;

class bitset {

public :
    bitset () ;
    bitset ( bitset & ) ;
    bitset ( int size , bool value = 0 ) ;
    ~ bitset () ;

    int get_size ( void ) ;     // get size
    void set_size ( int size ) ;    // set size
    void flip ( int bit ) ;     // bit = ! bit
    void set ( int bit ) ;      // bit = 1
    void reset_bit ( int bit ) ;    // bit = 0
    void reset ( void ) ;       // bitset = 0
    void print ( void ) ;       // print to screen
```

```
    bool operator [] ( int bit ) ;  // get bit
    bitset & operator += ( bitset & ) ;
    bitset & operator = ( bitset & ) ;

private :
    typedef long unsigned int block ;

    block * blocks ;
    int size_in_bits ;

    int num_blocks ( int ) ;
    int block_of_bit ( int ) ;
    int weight ( int ) ;
} ;

ostream & operator << ( ostream & , bitset & ) ;

# endif

/* End bitset.h */
```

```
/*************************************************************
 * bitset.cc                                                 *
 *      bitset object type and operations needed by          *
 *      Pomerance's Quadratic Sieve Algorithm                *
 *                                                           *
 *      Courtesy of                                          *
 *          Mike Roig <rtx@papelmail.com>                    *
 *      in collaboration with                                *
 *          R. Arigita <rou@papelmail.com>                   *
 *                                                           *
 *      Canterbury, UK, Feb 2003                             *
 *************************************************************/

# include "bitset.h"
# include <iostream>

using std :: ostream ;

// Constructors

bitset :: bitset () : size_in_bits( 0 ) , blocks ( ( block * ) 0 ) { }

bitset :: bitset ( bitset & rhs ) {
    set_size( rhs.get_size() ) ;
    for ( int i = 0 ; i < num_blocks ( get_size() ) ; ++ i )
        blocks[i] = rhs.blocks[i] ;
}

bitset :: bitset ( int size , bool value ) : size_in_bits( size ) {
    blocks = new block [ num_blocks( size ) ] ;
    for ( int i = 0 ; i < num_blocks( size ) ; ++ i )
        blocks[i] = value ? ~ (block) 0 : (block) 0 ;
}

// Destructor

bitset :: ~ bitset () {
    if ( blocks ) delete [] blocks ;
}

// Public operators
```

```
int bitset :: get_size ( void ) {
    return size_in_bits ;
}

void bitset :: set_size ( int size ) {
    if ( blocks ) delete [] blocks ;
    blocks = new block [ num_blocks( size ) ] ;
    size_in_bits = size ;
    for ( int i = 0 ; i < num_blocks( size ) ; ++ i )
        blocks[i] = 0 ;
}

void bitset :: flip ( int bit ) {
    blocks [ block_of_bit( bit ) ] ^= weight ( bit ) ;
}

void bitset :: set ( int bit ) {
    blocks [ block_of_bit( bit ) ] |= weight ( bit ) ;
}

void bitset :: reset_bit ( int bit ) {
    blocks [ block_of_bit( bit ) ] &= ~ weight ( bit ) ;
}

void bitset :: reset ( void ) {
    for ( int i = 0 ; i < num_blocks( get_size() ) ; ++ i )
        blocks[i] = 0 ;
}

void bitset :: print ( void ) {
    std :: cout << (* this) ;
    std :: cout << '\n' ;
}

bool bitset :: operator [] ( int bit ) {
    block * b = blocks + block_of_bit( bit ) ;
    bool r = ( (*b) & weight( bit ) ) ;
    return r ;
}

bitset & bitset :: operator = ( bitset & rhs ) {
    for ( int i = 0 ; i < num_blocks( get_size() ) ; i++ ) {
```

```
        blocks[ i ] = rhs.blocks[ i ] ;
    }
    return * this ;
}

bitset  & bitset :: operator += ( bitset & rhs ) {
    for ( int i = 0 ; i < num_blocks( get_size() ) ; i++ )
        blocks[ i ] ^= rhs.blocks[ i ] ;
    return * this ;
}

ostream & operator << ( ostream & str , bitset & m ) {
    int bitset_size = m.get_size() ;
    for ( int i = 0 ; i < bitset_size ; ++ i )
        str << m[i] ;
}

// Private operators

int bitset :: block_of_bit ( int bit ) {
    return ( bit / ( sizeof( block ) * 8 ) ) ;
}

int bitset :: weight ( int bit ) {
    return ( 1 << bit ) ;
}

int bitset :: num_blocks ( int size ) {
    static bool extra_block = ( size % ( sizeof(block)*8 ) ? 0 : 1 ) ;
    return ( size / sizeof( block ) * 8 + extra_block ) ;
}

/* End bitset.cc */
```

B.8.2 Screenshot

```
[r@localhost code]$ g++ -ggdb -c Integer.cc
[r@localhost code]$ g++ -ggdb -o QS.cc Integer.o bitset.cc -lgmp
[r@localhost code]$ ./QS 10403

Pomerance's Quadratic Sieve Factoring Algorithm
-----------------------------------------------
```

```
Entering Init phase...
    Number to factor: 10403
    [ sqrt( key ) ] = 101
    Building factor base...done!
    looking for roots of N mod p...done!
...Init phase done!

Building exponent matrix...
    Total f(x)'s needed: 9
    Sieving... f(x)'s factorized: 9
...exponent matrix done!

Entering Reduce phase... Factors: 101 x 103

Total Running time: 0s 113622mcs

[r@localhost code]$ ./QS 10403000001

Pomerance's Quadratic Sieve Factoring Algorithm
-----------------------------------------------

Entering Init phase...
    Number to factor: 10403000001
    [ sqrt( key ) ] = 101995
    Building factor base...done!
    looking for roots of N mod p...done!
...Init phase done!

Building exponent matrix...
    Total f(x)'s needed: 24
    Sieving... f(x)'s factorized: 24
...exponent matrix done!

Entering Reduce phase... Factors: 9 x 1155888889

Total Running time: 0s 961532mcs

[r@localhost code]$ ./QS 1040300000123

Pomerance's Quadratic Sieve Factoring Algorithm
-----------------------------------------------
```

```
Entering Init phase...
    Number to factor: 1040300000123
    [ sqrt( key ) ] = 1019950
    Building factor base...done!
    looking for roots of N mod p...done!
...Init phase done!

Building exponent matrix...
    Total f(x)'s needed: 33
    Sieving... f(x)'s factorized: 33
...exponent matrix done!

Entering Reduce phase... Factors: 47630603 x 21841

Total Running time: 172s 94769mcs

[r@localhost code]$ ./QS 10403000001234023470234094207411111

Pomerance's Quadratic Sieve Factoring Algorithm
-----------------------------------------------

Entering Init phase...
    Number to factor: 104030000012340234702340942071111
    [ sqrt( key ) ] = 10199509792746915
    Building factor base...done!
    looking for roots of N mod p...done!
...Init phase done!

Building exponent matrix...
    Total f(x)'s needed: 539
    Sieving... f(x)'s factorized: 0 Killed

[r@localhost code]$ exit
```

B.9 Lenstra's ECM Algorithm

```
/*************************************************************
 *  Lenstra.cc                                               *
 *    Uses Lenstra algorithm to factorize a large Integer    *
 *                                                           *
 *  Compile:                                                 *
 *    g++ -g -o Lenstra Lenstra.cc Integer.o -lgmp           *
 *                                                           *
 *  Run:                                                     *
 *    ./Lenstra <number_to_factor> [ Initial K ]             *
 *************************************************************/

#include <sys/time.h>
#include "Integer.h"

#define A_INCR  30
#define MAX_As  (A_INCR * 40)   // changes before making new curve
#define START_X  1              // initial x coordinate
#define START_Y  1              // initial y coordinate
#define DISC(a,b) (Integer) ( (4*a*a*a) + (27*b*b) ) // discriminant

Integer get_K( Integer ) ;
Integer get_factor( Integer, Integer ) ;

int main( int argc , char * argv[] ) {

    // Structures to control initial,
    // final and total running time
    struct timeval * tv1 = new timeval ;
    struct timeval * tv2 = new timeval ;
    struct timeval * tvt = new timeval ;
    struct timezone * tz = NULL ;

    Integer key, factor, K ;

    // get start time
    gettimeofday(tv1,tz);

    cout << "\n\tLenstra's Factoring Algorithm\n" ;
    cout << "\t----------------------------\n" << flush ;
```

```
    if ( argc < 2 ) {
        cerr << "Usage: " << argv[ 0 ]
                          << " <number_to_factor> [ Initial K ]" << endl ;
        cerr << "Example: " << argv[ 0 ] << " 1715761513 25" << endl ;
        return -1 ;
    }

    key = argv[ 1 ] ;
    if ( argc == 3 ) K = argv[ 2 ] ;
    else K = get_K( key ) ;            // default K if none given

    cout << "Number to factor: " << key << endl ;
    cout << "Factors found: " << flush ;

    if ( isprime( key ) ) {
        cout << key << " is prime !" << endl ;
        return 0 ;
    }

    // main loop
    while ( ! isprime( key ) ) {
        factor = get_factor( key , K ) ;
        cout << factor << " x " << flush ;
        key /= factor ;
        if ( argc != 3 ) K = get_K( key ) ;
    }

    cout << key << endl ;

    // get finish time
    gettimeofday(tv2,tz);
    // get total time
    timersub(tv2,tv1,tvt);
    cout << "Total Running time: " << tvt->tv_sec << "s " ;
    cout << tvt->tv_usec << "mcs\n\n" ;

    return 0 ;
}

Integer get_K( Integer n ) {
    Integer K ;
    K = (int) logn( n , 10 ) ;
```

```
    K = K * K / 2 ;
    if ( K < 2 ) K = 2 ;
    return K ;
}

Integer get_factor( Integer key , Integer K ) {

    Integer a, b, k, g, x, y ;
    Integer tmp_x, tmp_y, sum_x, sum_y, grad;
    int bits ;
    bool newcurve ;

    if ( key % 2 == 0 ) return 2 ;
    if ( key % 3 == 0 ) return 3 ;
    if ( key % 5 == 0 ) return 5 ;
    if ( key % 7 == 0 ) return 7 ;

    a = 1 ;
    k = LCM( K ) ;
    bits = log2( k ) ;

    for ( ; ; ) {

        x = START_X ;
        y = START_Y ;
        b = ( y * y ) - ( x * x * x ) - ( a * x ) ;

        // Make a valid curve
        g = gcd( DISC( a , b ) , key ) ;
        while ( g != 1 ) {
            if ( g < key ) {
                if ( ! isprime( g ) )
                    return get_factor( g , K ) ;
                else return g ;
            }
            else {
                a += A_INCR ;
                b = ( y * y ) - ( x * x * x ) - ( a * x ) ;
                g = gcd( DISC( a , b ) , key ) ;
            }
        }
```

```
sum_x = 0 ;
sum_y = 0 ;
newcurve = false ;

if ( a < MAX_As ) {

    for ( int i = 1 ; i <= bits ; i ++ ) {  // compute k*(x, y)
                        // using powers of 2
        tmp_x = x ;
        tmp_y = 2 * y ;
        g = gcd( tmp_y , key ) ;

        if ( 1 < g && g < key ) {
            if ( ! isprime( g ) )
                return get_factor( g , K ) ;
            else return g ;
        }
        if ( g == key ) {
            newcurve = true ;
            break ;
        }

        grad = InvModN( tmp_y , key ) * ( 3 * ( x * x ) + a ) ;
        x = ( grad * grad - 2 * x ) % key ;
        y = ( grad * ( tmp_x - x ) - y ) % key ;

        if ( ! testbit( k , i ) )
            continue ;
        if ( ! sum_y ) {
            sum_x = x ;
            sum_y = y ;
            continue ;
        }

        //Need to add {x,y} in to sum_{x,y}
        tmp_x = ( x - sum_x ) % key ;
        tmp_y = ( y - sum_y ) % key ;
        g = gcd( tmp_x , key ) ;
        if ( 1 < g && g < key ) {
            if ( ! isprime( g ) )
                return get_factor( g , K ) ;
            else return g ;
```

```
                }
                if ( g == key ) {
                    newcurve = true ;
                    break ;
                }
                grad = InvModN( tmp_x , key ) * tmp_y ;
                tmp_x = ( ( grad * grad - sum_x ) - x ) % key ;
                tmp_y = ( grad * ( sum_x - tmp_x ) - sum_y ) % key ;
                sum_x = tmp_x ;
                sum_y = tmp_y ;
            }
        }
        else newcurve = true ;

        if (newcurve) {
            while ( k == LCM(K) ) K ++ ;
            a = 1 ;
            k = LCM(K) ;
            bits = log2(k) ;
        }
        else a += A_INCR ;
    }
}

/* <rou@papelmail.com> */
```

B.9.1 Screenshot

```
[r@localhost code]$ g++ -ggdb -c Integer.cc
[r@localhost code]$ g++ -ggdb -o ECM Lenstra.cc Integer.o -lgmp
[r@localhost code]$ ./ECM 555555551

    Lenstra's Factoring Algorithm
    -----------------------------
Number to factor: 555555551
Factors found: 7417 x 74903
Total Running time: 0s 68021mcs

[r@localhost code]$ ./ECM 555555551555551

    Lenstra's Factoring Algorithm
    -----------------------------
```

```
Number to factor: 555555551555551
Factors found: 555555551555551 is prime !

[r@localhost code]$ ./ECM 55555555155555111

    Lenstra's Factoring Algorithm
    -----------------------------
Number to factor: 55555555155555111
Factors found: 3 x 1361 x 163 x 32983 x 2530873
Total Running time: 0s 343881mcs

[r@localhost code]$ ./ECM 5555555515555511111111

    Lenstra's Factoring Algorithm
    -----------------------------
Number to factor: 55555555155555111111111
Factors found: 55555555155555111111111 is prime !

[r@localhost code]$ ./ECM 555555551555555111111122221321

    Lenstra's Factoring Algorithm
    -----------------------------
Number to factor: 555555551555555111111122221321
Factors found: 13 x 2267 x 188509229939788643449873 51
Total Running time: 0s 359977mcs

[r@localhost code]$ ./ECM 5555555515555511111111222272333333

    Lenstra's Factoring Algorithm
    -----------------------------
Number to factor: 5555555515555511111111222272333333
Factors found: 31 x 17 x 1153 x 2924921 x 170483 x 18335482780196801
Total Running time: 0s 682020mcs

[r@localhost code]$ exit
```

B.10 Makefile

```
#########################################################
#  Factorization Algorithms, Rodrigo Arigita, Feb 2003  #
#########################################################

OPTIONS = -ggdb -Wno-deprecated

all : Trial Fermat Shanks Pollard_rho Pollard_p-1 QS Lenstra

Trial : Trial_division.cc Integer.o
    g++ $(OPTIONS) -o Trial Trial_division.cc Integer.o -lgmp

Fermat : Fermat.cc Integer.o
    g++ $(OPTIONS) -o Fermat Fermat.cc Integer.o -lgmp

Shanks : Shanks.cc Integer.o
    g++ $(OPTIONS) -o Shanks Shanks.cc Integer.o -lgmp

Pollard_rho: Pollard_rho.cc Integer.o
    g++ $(OPTIONS) -o Pollard_rho Pollard_rho.cc Integer.o -lgmp

Pollard_p-1: Pollard_p-1.cc Integer.o
    g++ $(OPTIONS) -o Pollard_p-1 Pollard_p-1.cc Integer.o -lgmp

QS: QS.cc bitset.cc Integer.o
    g++ $(OPTIONS) -o QS QS.cc bitset.cc Integer.o -lgmp

Lenstra: Lenstra.cc Integer.o
    g++ $(OPTIONS) -o Lenstra Lenstra.cc Integer.o -lgmp

Integer.o : Integer.cc Integer.h
    g++ $(OPTIONS) -c Integer.cc

clean :
    rm *.o Trial Fermat Shanks Pollard_rho Pollard_p-1 QS Lenstra
```

B.10.1 Screenshot

```
[r@localhost code]$ ls -l
total 76
-rwxrwxr-x    1 r        r            3159 Mar 22 14:06 bitset.cc
-rwxrwxr-x    1 r        r            1626 Mar 22 14:06 bitset.h
-rwxrwxr-x    1 r        r            2445 Mar 22 14:06 Fermat.cc
-rwxrwxr-x    1 r        r           12390 Mar 22 14:06 Integer.cc
-rwxrwxr-x    1 r        r            8048 Mar 22 14:06 Integer.h
-rwxrwxr-x    1 r        r            4748 Mar 22 14:06 Lenstra.cc
-rwxrwxr-x    1 r        r            1039 Mar 22 14:06 Makefile
-rwxrwxr-x    1 r        r            2414 Mar 22 14:06 Pollard_p-1.cc
-rwxrwxr-x    1 r        r            2586 Mar 22 14:06 Pollard_rho.cc
-rwxrwxr-x    1 r        r            7674 Mar 22 14:06 QS.cc
-rwxrwxr-x    1 r        r            2755 Mar 25 04:02 Shanks.cc
-rwxrwxr-x    1 r        r            2464 Mar 22 14:06 Trial_division.cc
[r@localhost code]$ make
g++ -ggdb -Wno-deprecated -c Integer.cc
g++ -ggdb -Wno-deprecated -o Trial Trial_division.cc Integer.o -lgmp
g++ -ggdb -Wno-deprecated -o Fermat Fermat.cc Integer.o -lgmp
g++ -ggdb -Wno-deprecated -o Shanks Shanks.cc Integer.o -lgmp
g++ -ggdb -Wno-deprecated -o Pollard_rho Pollard_rho.cc Integer.o -lgmp
g++ -ggdb -Wno-deprecated -o Pollard_p-1 Pollard_p-1.cc Integer.o -lgmp
g++ -ggdb -Wno-deprecated -o QS QS.cc bitset.cc Integer.o -lgmp
g++ -ggdb -Wno-deprecated -o Lenstra Lenstra.cc Integer.o -lgmp
[r@localhost code]$ ls -l
total 2140
-rwxrwxr-x    1 r        r            3159 Mar 22 14:06 bitset.cc
-rwxrwxr-x    1 r        r            1626 Mar 22 14:06 bitset.h
-rwxr-xr-x    1 r        r          248907 Mar 26 00:47 Fermat
-rwxrwxr-x    1 r        r            2445 Mar 22 14:06 Fermat.cc
-rwxrwxr-x    1 r        r           12390 Mar 22 14:06 Integer.cc
-rwxrwxr-x    1 r        r            8048 Mar 22 14:06 Integer.h
-rw-r--r--    1 r        r          214312 Mar 26 00:47 Integer.o
-rwxr-xr-x    1 r        r          269903 Mar 26 00:48 Lenstra
-rwxrwxr-x    1 r        r            4748 Mar 22 14:06 Lenstra.cc
-rwxrwxr-x    1 r        r            1039 Mar 22 14:06 Makefile
-rwxr-xr-x    1 r        r          248912 Mar 26 00:47 Pollard_p-1
-rwxrwxr-x    1 r        r            2414 Mar 22 14:06 Pollard_p-1.cc
-rwxr-xr-x    1 r        r          251375 Mar 26 00:47 Pollard_rho
-rwxrwxr-x    1 r        r            2586 Mar 22 14:06 Pollard_rho.cc
-rwxr-xr-x    1 r        r          330038 Mar 26 00:48 QS
```

```
-rwxrwxr-x    1 r        r            7674 Mar 22 14:06 QS.cc
-rwxr-xr-x    1 r        r          255531 Mar 26 00:47 Shanks
-rwxrwxr-x    1 r        r            2755 Mar 25 04:02 Shanks.cc
-rwxr-xr-x    1 r        r          247927 Mar 26 00:47 Trial
-rwxrwxr-x    1 r        r            2464 Mar 22 14:06 Trial_division.cc
[r@localhost code]$ ./Trial

    Trial Division Factoring Algorithm
    ----------------------------------
    Usage: ./Trial <number_to_factor>

[r@localhost code]$ ./Fermat

    Fermat's Factoring Algorithm
    ----------------------------
    Usage: ./Fermat <number_to_factor>

[r@localhost code]$ ./Shanks

    Shank's SQUFOF Factoring Algorithm
    ----------------------------------
    Usage: ./Shanks <number_to_factor>

[r@localhost code]$ ./Pollard_rho

    Pollard rho Factoring Algorithm
    -------------------------------
    Usage: ./Pollard_rho <number_to_factor>

[r@localhost code]$ ./Pollard_p-1

    Pollard p-1 Factoring Algorithm
    -------------------------------
    Usage: ./Pollard_p-1 <number_to_factor>

[r@localhost code]$ ./QS

    Pomerance's Quadratic Sieve Factoring Algorithm
    -----------------------------------------------
    Usage: ./QS <number_to_factor>

[r@localhost code]$ ./Lenstra
```

```
    Lenstra's Factoring Algorithm
    -----------------------------
    Usage: ./Lenstra <number_to_factor> [ Initial K ]
    For an example: ./Lenstra 1715761513 25

[r@localhost code]$ exit
```

All source code presented here has been carefully designed and debuged with *gdb* as a good exercise and experience with the C++ programming language. I haven't got the time nor the resources to test the code on any other machine than mine, which I described at the beginning of the Appendix. I think it should work on any other architecture appart from my i586. Minor modifications can improve considerably the algorithms.

B.11 Sunshine

the smell of sunshine
I remember sometimes
I've done all I can do
can I please come with you?
sweet smell of sunshine
I remember sometimes[1]

[1] *I'm looking forward to joining you, finally. T. Reznor, NIN, The Fragile, right*

Bibliography

[1] H. Riesel, *Prime Numbers and Computers Methods for Factorization.* (Birkhäuser)

[2] D.M. Bressoud, *Factorization and Primality Testing.* (Springer-Verlag)

[3] Carl Pomerance, *Cryptology and Computational Number Theory.* (American Mathematical Society, Vol.42)

[4] Neal Koblitz, *A Course in Number Theory and Cryptography.* (Springer-Verlag)

[5] Donald E. Knuth, *The Art of Computer Programming, Vol.2, Seminumerical Algorithms.* (Addison-Wesley)

[6] M.A. Morrison and J. Brillhart, *A Method of Factorization and the Factorization of* F_7. (Math. Comp. Vol. 29 (1975), p.183-205)

[7] J.L. Gerver, *Factoring Large Numbers with a Quadratic Sieve.* (Math. Comp. Vol. 41 (1983), p.287-294)

[8] J.D. Dixon, *Asymptotically Fast Factorization of Integers.* (Math. Comp. Vol. 36 (1981), p.255-260)

[9] C.P. Schnorr, H.W. Lenstra, Jr., *A Monte Carlo factoring algorithm with linear storage.* (Math. Comp. Vol. 43 (1984), p.289-311)

[10] P.L. Montgomery, *Speeding the Pollard and Elliptic curve Methods of Factorization.* (Math. Comp. Vol. 48 (1987), p.243-264)

[11] P.L. Montgomery, R.D. Silverman, *An FFT extension to the* $p-1$ *factoring algorithm.* (Math. Comp. Vol. 54 (1990), p.839-854)

[12] James A. Davis, Diane B. Holdridge, *Factorization of large integers on a massively parallel computer.* (Advances in Cryptology - EUROCRYPT '88)

[13] A.K. Lenstra, M.S. Manasse, *Factoring with two large primes.* (Advances in Cryptology - EUROCRYPT '90)

[14] J.M. Pollard, *A Monte Carlo Method for Factorization.*
(BIT 15 (1975), p.331-334.)

[15] Richard P. Brent, *An improved Monte Carlo Factorization Algorithm.*
(BIT 20 (1980), p.176-183)

[16] Olof Åsbrink, Joel Brynielsson, *Factoring Large Integers using Parallel Quadratic Sieve.* (April 2000)

[17] Eric Landquist, *The Quadratic Sieve Factoring Algorithm.*
(MATH 488 (2001): Cryptographic Algorithms)

[18] Robert Silverman, *The Multiple Polynomial Quadratic Sieve Method of Computation.*
(Math. Comp. Vol. 48 (1987), p.329-340)

[19] A.K. Lenstra, H.W. Lenstra, Jr., *The development of the Number Field Sieve.*
(Springer-Verlag, 1993)

[20] H.W. Lenstra, Jr., *Factoring integers with Elliptic Curves.*
(Ann. of Math. (2) 126 (1987), p.649-673)

[21] Richard P. Brent, *Some Integer Factorization Algorithms using Elliptic Curves.*
(CS Laboratory, Australian National University, 1998)

[22] J. H. Silverman, J. Tate, *Rational Points on Elliptic Curves.*
(Springer-Verlag)

[23] A.O.L. Atkin, F. Morain, *Finding suitable curves for the Elliptic Curve method of factorization.* (Math. Comp. Vol. 60 (1995), p.399-405)

[24] Peter W. Shor, *Introduction to Quantum Algorithms.*
(AT&T Research, New Jersey. 2001)

[25] Peter W. Shor, *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer.* (AT&T Research, New Jersey. 1996)

[26] T. Back, D.B. Fogel, Z. Michalewicz, *Handbook of Evolutionary Algorithms.*

[27] R. Arigita, *Quantum Momentum.*
(CO619 Assessment 3, March 2003)

[28] Caxton C. Foster, *Cryptanalysis for Microcomputers.*
(Haiden)

[29] Bjarne Stroustrup, *The C++ programming language.*
(3^{rd} edition, Addison Wesley)