Ferienakademie 2001:
Cryptography and Security of Open Systems

Factorization of Large Integers

Stefan Büttcher
Sieglitzhofer Str. 19, 91054 Erlangen

<stefan@buettcher.org>

September 22, 2001

Contents

Sources of a Java Factorization class, using all the algorithms presented in this essay, can be found at http://stefan.buettcher.org/cs/factorization/ .

1 Introduction and Overview

Modern public-key cryptographic systems are based on modular arithmetic. The security of many of them relies on the intractability of the factorization problem, i.e. the difficulty to compute the prime factors (or any non-trivial factor at all) of a given integer. The most prominent examples of these algorithms are the RSA cryptosystem ([RSA1978]) and the Rabin public-key encryption scheme ([RAB1979]).

Definition. Let P be the set of all primes,

$$P := \{i \in N : p \text{ is prime}\}, \text{and } \hat{P} := \{(p,\ n) : p \in P, n \in N\}$$

the set of all prime powers, further

$$P_0 := \{M \subset \hat{P} : (p,\ j) \in M \Rightarrow (\nexists\ k \neq j : (p,k) \in M)\},$$

i.e. the set of all sets of prime powers where the same prime appears at most once. The Fundamental Theorem of Arithmetic allows to construct any positive integer by means of the following function

$$mult : P_0 \rightarrow N;\ mult(M) = \prod_{(j,p)\in M} p^j,$$

which takes a set of prime powers and returns the integer that is the product of these powers. Then, the factorization of any positive integer can be defined as

$$fact : N \rightarrow P_0;\ fact(mult(M)) = M \quad \forall\ M \in P_0.$$

While the value of $mult(M)$ can easily be computed by multiplying the primes given with their respective powers, its reverse operation $fact(mult(M))$, i.e. the decomposition, is likely to be more difficult to compute. Certainly, it can be computed and it can be done using finite time and space, but the amount of time and/or space needed for doing so dramatically explodes as the length of the number to be decomposed increases.

After a short review of the RSA cryptosystem, some ways of computing the decomposition of a given integer will be shown. Starting from the early attempts, *Trial Division* and *Fermat's Algorithm*, *Pollard's Rho* and *p-1* methods will be explained, which do an excellent job on integers with certain properties. Eventually, two modern probabilistic algorithms will be presented that are based on quadratic forms: Dixon's algorithm with Pomerance's improvement, leading to the *Quadratic Sieve*, and the Brillhart-Morrison *Continued Fraction Algorithm* (CFRAC).

2 Review of the RSA Cryptosystem

The RSA Cryptosystem uses computations in $\mathcal{Z}_n$ where $n = pq$ is a product of two distinct primes. Every RSA configuration can be described as a tuple

$$K := (n, p, q, a, b) :\ n = pq,\ ab \equiv 1 \pmod{\phi(n)}$$

with $\phi(n) = (p-1)(q-1)$ being Euler's Phi function. Any (plaintext) number $x \in \mathcal{Z}_n$ can now be encrypted by performing the operation

$$encrypt_K(x) := x^b \bmod n,$$

yielding the (ciphertext) number y. Decryption is done by computing

$$decrypt_K(y) := y^a \bmod n,$$

where a is the user-chosen private key, while b is the publicly available public key. The values p, q and a must be kept secret in order to guarantee the security of RSA, while the product $n = pq$ may (and must) be publicly available.

More information on RSA and related topics as well as some deeper analysis can be found in [SCH1996], [MOV1996] and [STI1995].

2.1 Correctness of RSA

RSA being correct means that encryption and decryption are indeed inverse operations, which can easily be proven. Since

$$a \times b \equiv 1 \pmod{\phi(n)},$$

there is an integer $t \geq 1$, so that

$$a \times b = t \times \phi(n) + 1.$$

For any plaintext $x \in \mathcal{Z}_n$ it is $decrypt_K(encrypt_K(x)) \equiv (x^b)^a \pmod n$.

$$\begin{aligned}(x^b)^a &\equiv x^{ba} \pmod n \\ &\equiv x^{t\phi(n)+1} \pmod n \\ &\equiv (x^{\phi(n)})^t \times x \pmod n \\ &\equiv x \pmod n.\end{aligned}$$

The last – and essential – congruence directly arises from Euler's Theorem, which states that

$$a^{\phi(m)} \equiv 1 \pmod m$$

for any a that is relatively prime to m [BUN1998, pp. 96,97].

2.2 The RSA Problem and its Relationship to Factorization and the Discrete Logarithm

Definition. Let $K := (n, p, q, a, b)$ be an RSA configuration with the usual properties. The task to compute the private key a while only knowing n and the public key b is called the *RSA Problem*.

While it is uncertain yet if the RSA Problem and the Factorization Problem are computationally equivalent, it is obvious that the RSA Problem can be reduced to Factorization using a polytime reduction function. This means that if one is able to decompose n to its prime factors p and q, then he or she is also able to compute a from the public key b. A reduction in the opposite direction still remains to be shown [MOV1996, p. 99]. There are, however, interesting approaches in [BUC2001, p. 119] and [STI1995, p. 139], which use probabilistic methods for calculating the decomposition when only knowing a, b and n.

RSA Problem $\leq_p$ *Factorization Problem*

Intuitively, this must be true, since calculating a from b is no different from calculating b from a (due to the symmetry $ab \equiv 1 \pmod{n}$), which the creator of the RSA configuration in question of course has done. In fact, knowing the secret prime factors p and q, the private key a can be computed in polytime using the Extended Euclidean Algorithm [STI1995, p. 119].

Factorization Problem $\leq_p$ *Discrete Logarithm Problem*

Another interesting thing to show is that the Factorization Problem can be reduced to the Discrete Logarithm Problem: Assume there is a program $\mathcal{P}$ that is able to compute the exponent e with respect to a base b in polytime, so that

$$b^e \equiv x \pmod{n}$$

for any given integer x, i.e.that

$$b^{\mathcal{P}(\mathrm{b,x})} \equiv x \pmod{n}.$$

Since

$$(x^b)^a \equiv x \pmod{n},$$

it is possible to launch a chosen-plaintext attack by selecting any plaintext x and computing the corresponding ciphertext x^b. Applying $\mathcal{P}$ then reveals the private key a:

$$a = \mathcal{P}(x^b, x),$$

which effectively means that the RSA Problem can be solved in polytime. This gives some motivation to study the Discrete Log Problem and to look for some efficient ways of computation. That is, however, far beyond the scope of this essay, which focuses on Factorization only.

Further Information on the Discrete Logarithm Problem can be found reading the usual suspects: [STI1995, pp. 162-177] and [MOV1996, pp. 103-112].

Another interesting approach to Integer Factorization (and the Discrete Log) was suggested in 1994 by [SHO1994]. It is based on quantum computers and their theoretical ability to solve any NP-problem in polytime due to their non-deterministic nature.

3 Basic Thoughts and First Steps: Trial Division

Solving the Factorization Problem for a given integer seems to be a difficult or at least expensive task, while deciding if that integer is prime or composite is much easier and very fast using reliable probabilistic methods such as Fermat's "little" Theorem or even more sophisticated algorithms. Hence, before starting the factorization, it is always a good idea to check the number to be factored for primality. While this is a general recipe for any factorization method, it is even more important for the probabilistic algorithms. Probabilistic factorization means that the algorithm does not necessarily terminates, even if it is run on a composite number. So, it is impossible to distinguish between bad luck and primality when trying to factor a really big integer. Information on Primality Testing is given in [MOV1996], [GAR1998] and [BRE1989].

The most obvious way of decomposing a composite integer n is trying to divide n by all positive integers $k \leq \sqrt{n}$. This procedure is called *Trial Division*. While the memory necessary for running the Trial Division is $O(|n|)$ (where $|n|$ stands for the length of n's binary representation), its time consumption in most cases makes it impossible to be used for finding factors: Assuming a division algorithm whose time consumption is linear in $|n|$ (which is impossible, but that does not matter right now), the running time of the naive Trial Division is about $\Theta(|n| \times \sqrt{2}^{|n|})$.

There are some possibilities of speeding up the Trial Division. The most popular way is the use of the Sieve of Eratosthenes (cf. [BUN1998, p. 286]). The sieving algorithm can be used to build up a prime database before launching the actual Trial Division. By this, only the primes $p \leq \sqrt{n}$ need to be examined for their ability to divide into n.

```
Number trialDivision(Number n) {
  Number s := sqrt(n);
  boolean isPrime[] := new boolean[s];
  for (i = 2; i <= s; i++)
    isPrime[i] := true;
  for (i = 2; i <= sqrt(s); i++)
    if (isPrime[i]) {
      j := i * i;
      while (j <= s) {
        isPrime[j] := false;
        j := j + 2;
      }
    }
  for (i = 2; i <= s; i++)
    if ((isPrime[i]) && (n % i == 0))
      return i;
  return -1;
}
```

Algorithm 3.1: Trial Division with the Sieve of Eratosthenes.

As the studies of primes have shown, the number of primes lower than a given bound x is $\pi(x)$ which can be approximated by $\frac{x}{\ln x}$. Thus, at least $\frac{\sqrt{n}}{\ln \sqrt{n}}$ distinct trial divisions are necessary to show that n is prime. Since the prime numbers of a given RSA modulus usually are of roughly the same size, approximately that many ($\frac{\sqrt{n}}{\ln \sqrt{n}}$) trial divisions are needed as well to decompose that number. Using a division algorithm that is linear in the length of number to be divided, as supposed above, will take $\Theta(\sqrt{2}^{|n|})$, which is way better than the naive approach, but still highly exponential in $|n|$.

Therefore, Trial Division can be applied to small integers, say up to 12^{14}, as they definitely have at least one prime factor $p \leq \sqrt{n} = 10^7$. Since

$$\pi(10^7) \approx \frac{x}{\ln x} = 620,000 \ ,$$

no more than 620,000 divisions are required in order to find a divisor of the integer in question (worst-case scenario). Probably, a prime factor can be found much ear-

lier.

Besides factoring small integers, due to its amazing pace when limiting the number of primes to be examined, Trial Division is often used to extract all small prime factors p_i from a given number n and then continue to factor the remaining quotient

$$\hat{n} = \frac{n}{\prod p_i}.$$

This can be done using some more sophisticated algorithms that suit large numbers much better than Trial Division does. As it does not cost too much, it is always recommended to spend a few seconds in Trial Division before passing over to the other algorithms.

4 Factoring integers with certain properties

Algorithm 3.1 uses some factor database to decompose a given integer n to its prime divisors p_i. However, although the main target is to find the prime factors of n, it is in fact not necessary to compute them directly. Instead, it is sufficient just to find any integer $m : m|n$. m can then be checked for primality: If it is prime, it has to be continued with the decomposition of $\frac{n}{m}$. Otherwise, m is to be decomposed and after that has been done, one can go on with $\frac{n}{m}$. This leads to the following algorithm, which eventually reveals all the prime factors of n.

```
void findPrimeFactors(Number n, Vector factors) {
  if (isPrime(n))
    factors.add(n);
  else {
    m = findFactor(n);
    findPrimeFactors(m, factors);
    findPrimeFactors(n.divide(m), factors);
  }
}
```

Algorithm 4.1: Successive factorization of a given integer.

4.1 Fermat's Algorithm

One of the first algorithms in history that works fine for integers with special properties is Fermat's Algorithm, invented by the Pierre de Fermat. It works excellent on integers n whose prime factors are close together, i.e. relatively close to $\sqrt{n}$. Fermat's method is not in fashion anymore but is interesting anyway, because it can show the basic idea that most of today's fashionable algorithms are based on. The idea is the following. Imagine the number n to be factored can be written as

$$\begin{aligned} n &= x^2 - y^2 \\ &= (x-y) \times (x+y). \end{aligned}$$

Then two non-trivial factors of n have been found. Furthermore, if n is odd – which we can certainly assume, since all powers 2^i have been removed by Trial Division –

and composite, then n can always be represented in that way. Let

$$n = a \times b\ , \quad x := \frac{a+b}{2} \ \text{ and } \ y := \frac{a-b}{2}.$$

Note that since n is odd, a and b are odd as well, which means that $a+b$ and $a-b$ are even and these equations make sense. Then we have got

$$x^2 - y^2 = \frac{a^2 + 2ab + b^2}{4} - \frac{a^2 - 2ab + b^2}{4} = a \times b = n.$$

For any odd integer n to be factored, Fermat's Algorithm starts with $x := \lceil \sqrt{n} \rceil$ and $y := 0$. It then increases y until $x^2 - y^2 \leq n$. If $(x^2 - y^2)$ is equal to n, the decomposition has been found, the algorithm halts. If it is lower than n, x is increased by 1 and the algorithm continues the y-loop.
Some additional definitions

$$\begin{aligned} r &:= x^2 - y^2 - n, \\ u &:= (x+1)^2 - x^2 = 2x + 1, \\ v &:= (y+1)^2 - y^2 = 2y + 1 \end{aligned}$$

lead to a streamlined form of Fermat's Algorithm presented in [BRE1989, p. 59] where it is only necessary to look for r's equality to zero:

```
s := sqrt(n) + 1;
u := 2 * s + 1;
v := 1;
r := s * s - n;
while (r != 0) {
  while (r > 0) {
    r := r - v;
    v := v + 2;
  }
  if (r < 0) {
    r := r + u;
    u := u + 2;
  }
}
a := (u + v - 2) / 2;
b := (u - v) / 2;
```

Algorithm 4.2: Fermat's Algorithm, streamlined.

When this algorithm has finished, a and b contain to divisors of n. Amazingly, due to the introduction of u and v, the main loop contains no multiplication or division at all (division by 2 is no real division, but can be implemented using a shift operation instead). Hence, it runs extremely quickly. Unfortunately, the number of loops required to compute the factors will grow very fast if a and b are not close to $\sqrt{n}$. In fact, due to the lack of any sieving mechanism, in the average case it is much slower than Trial Division, which turned out to be painfully slow.

Maurice Kraitchik (cf. [BRE1989, p. 61]) realized that a major saving of time could be achieved by replacing the equality

$$x^2 - y^2 = n$$

by the congruence

$$x^2 \equiv y^2 \pmod{n}.$$

Kraitchik looked for random integers x and y satisfying the congruence. Having found such a pair, it is clear that

$$\begin{aligned} & n \mid (x^2 - y^2) \\ \Rightarrow \quad & n \mid (x+y) \times (x-y). \end{aligned}$$

Now, chances are excellent that

$$n \not\mid (x-y),$$

which means that $gcd(n,\ x-y)$ is a non-trivial factor of n. The computation of the Greatest Common Divisor $gcd(\cdot,\cdot)$ of two integers can be performed using Euclid's Division Algorithm. The running time for computing $gcd(r,s)$ is $\mathcal{O}(|r| \times |s|)$ ([BUC2001, p. 17]). In other words: It can be done pretty fast. However, Kraitchik's method of finding such congruences, namely randomly taking a pair (x,y) and hoping it works, really needs some improvement, which will be presented in section 5. The basic idea, though, to find congruences $x^2 \equiv y^2 \pmod{n}$, remains the same.

4.2 Pollard p-1 and Pollard Rho

In 1974, John Pollard presented a new algorithm that is extremely efficient in factoring integers whose prime factors meet certain criteria ([BRE1989]). Let p be a prime divisor of n (the number to be factored) and $p-1$ be composed of prime factors q_i that are smaller than a given bound B. If it was possible to construct a number k so that $(p-1)|k$, then probably a non-trivial divisor of n would have been found:

Since

$$a^{p-1} \equiv 1 \pmod{p},$$

according to Fermat's Theorem, we have got

$$\begin{aligned} & a^k \equiv 1 \pmod{p} \\ \Rightarrow \quad & p \mid (a^k - 1). \end{aligned}$$

Chances are that $n \not\mid (a^k - 1)$. Then $gcd(n,\ a^k - 1)$ is a non-trivial divisor of n and we are happy :-). Only one single problem persists: How to find the mysterious exponent k?

Definition. Let B be a positive integer. An integer n is said to be $B-smooth$ if all prime factors p of the integer in question n satisfy the inequality $p \leq B$. If for all prime powers $p^{e_p} \mid n$ the inequality $p^{e_p} \leq B$ is satisfied as well, then n is called $B-powersmooth$ (cf. [MOV1996, p. 92]).

Now, imagine that $p-1$ is $10000-powersmooth$. Hence, we choose $k := 10000!$ and are done. Since modular exponentiation is so fast, the p-1 Algorithm can be performed very quickly. [MOV1996] states that the running time for finding the prime factor p is $\mathcal{O}(\frac{B \times |n|}{\ln(B)})$.

```
m := a;
for (j := 1; j <= B; i++) {
  m := m^k mod n;
  if (j % 5 == 0) {
    g = gcd(m - 1, n);
    if (g > 1) return g;
  }
}
```

Algorithm 4.3: Pollard's p-1 method.

As you can see, the algorithm does not simply rush through the $a^{j!}$ until $j! = k = B!$, which it could have done. This is due to two reasons: First, if j is bigger than all primes p dividing n, then nothing is won. $gcd(m-1, n) = n$ and no factor is found. Second, it is a waste of time. Most probably, the smoothness bound B is much lower than the argument passed to the function. Therefore, in every 5th cycle it is checked if a factor has been found. If so, the function returns it to the caller. Otherwise, it continues with exponentiation.

Further development of Pollard's p-1, combined with the theory of elliptic curves, resulted in the Elliptic Curves Method (ECM), one of the most powerful factorization methods in use today. ECM no longer depends on the prime composition of n.

On year after publishing his p-1 Algorithm, Pollard presented his next factorization algorithm, called Rho method. Due to its pseudo-random nature, Pollard first called his newly invented way a Monte Carlo Method but later changed its name to Rho – for reasons that will become obvious soon. Pollard Rho is excellent in finding prime factors that are too big for Trial Division but also cannot be efficiently found by the Quadratic Sieve, e.g. $p \approx 10^{10} - 10^{13}$.

Let n be a composite number, as before, and p a non-trivial divisor. Further, let $f(x) := x^2 + 1$. Beginning with any integer $x_0 \; \epsilon \; \mathcal{Z}_n$, consider the sequence $x_0,\ x_1,\ x_2,\ ...$, recursively defined by

$$x_{i+1} := f(x_i) \bmod n \quad \forall\ i \ \geq 0.$$

Let $y_i := x_i \bmod p$. Since

$$x_i \equiv f(x_{i-1}) \pmod{n},$$

it is clear that

$$y_i \equiv f(y_{i-1}) \pmod{p},$$

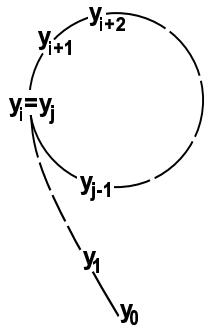
As there are only p distinct congruence classes modulo p, the sequence eventually comes to a point where

$$y_i = y_j$$

and thus

$$y_{i+k} = y_{j+k} \quad \forall\ k \; \epsilon \; N.$$

The sequence of y_i's now looks like a circle with a tail, i.e. the Greek letter Rho:



Furthermore, $y_i = y_j$ implies that

$$x_i \equiv x_j \pmod{p} \;\Rightarrow\; p \mid (x_i - x_j).$$

If $x_i \neq x_j$, then $gcd(n, x_i - x_j)$ is a non-trivial divisor of n.

The most obvious way of finding such a pair (x_i, x_j) is storing all values in a huge database. However, such a structure tends to become really huge as the length of the cycle increases. Therefore, its use is not recommended here.

R.P. Brent (cf. [BRE1989, p. 64]) made a different approach for finding such collisions. He suggested looking at the differences

$$x_1 - x_3,$$
$$x_3 - x_6,\; x_2 - x_7,$$
$$x_7 - x_{12},\; x_7 - x_{13},\; x_7 - x_{14},\; x_7 - x_{15},$$

more generally

$$x_{2n-1} - x_{22n-j}, 0 < j \leq 2n - 1.$$

Note that Brent's method not only examines every possible difference, but also makes sure that the search is done off the tail, since the tail is always shorter than the cycle.

The main idea that gives the Rho algorithm its impressive speed is not to look for every difference $x_i - x_j$ if it satisfies

$$1 < gcd(n, x_i - x_j) < n,$$

but to combine several such differences, say 10, and examine if their product satisfies

$$1 < gcd(n, \prod_{k=1}^{10} (x_i - x_{j_k}) < n.$$

This reduces the time needed enormously.

```
x1 := 2;
x2 := x1^2 + c;
cyclelen := 1;
product := 1;
done := 0;
while (done <= max) {
  for (j := 1; j <= cyclelen; j++) {
    x2 := x2^2 + c mod n;
    product := product * (x1 - x2) mod n;
```

```
      done++;
      if (done % 10 == 0) {
        g = gcd(n, product);
        if (g > 1) return g;
        product := 1;
      }
      x1 := x2;
      cyclelen := 2 * cyclelen;
      for (j := 1; j <= cyclelen; j++)
        x2 := x2^2 + c mod n;
    }
    g = gcd(n, product);
    if (g > 1) return g;
  }
```

Algorithm 4.4: Brent's version of Pollard's Rho method.

As you can see, the polynomial used to create the sequence is not fix but can be changed to $f(x) := x^2 + c$ for (almost) any c. The main point is that it is irreducible.

Example. Consider the number to be factored is $n = 7 \times 13 = 9$. Let the sequence be generated by $f(x) := (x^2 + 1) \bmod n$, resulting in

$$x_1 = 2,\ x_3 = 26,\ x_6 = 5,\ x_7 = 26.$$

Thus, after the first run of the j-loop, *product* is $x_1 - x_3 = -24$ and g is $gcd(n,\ 24) = 1$. After the second run, *product* is $(x_3 - x_6) \times (x_3 - x_7) = 0$. Bad luck! The reaction is simple: Take another c, e.g. $c := 3$. The resulting sequence is

$$x_1 = 2,\ x_3 = 52,\ x_6 = 17,\ x_7 = 19.$$

After the first loop, product is $x_1 - x_3 = -50$, thus g is 1. After the second loop, product is $(x_3 - x_6) \times (x_3 - x_7) \bmod n = 63$. g is $gcd(n,\ 63) = 7$, and a non-trivial divisor of $n = 91$ has been found.

Since the tail can be expected to be longer than just 2 or 3, it is recommended to start off with an i close to the expected length of the tail.

The number of cycles needed to find a divisor can be expected to be about $\sqrt{p_{min}}$, where p_{min} is the smallest prime divisor of n, the integer to be factored. The worst-case scenario, however, lets the algorithm perform p_{min} cycles before finding the divisor.

When trying to factor a typical RSA-modulus n, where $p \approx \sqrt{n}$ for both prime factors, the number of cycles needed can be expected to be $\frac{\lfloor n \rfloor}{4}$, each cycle containing 2 multiplications and 2 divisions.

5 The Quadratic Sieve

If a reliable primality test has revealed that a given integer n is composite, if it has been tried to decompose n by Trial Division in order to find the small factors, say $\leq 10^8$, then passed over to Pollard's methods of finding medium sized factors in the region of $10^{12}...10^{15}$ and there has still been no success, i.e. no non-trivial

divisor has been found, then time has come to show the power of QS to that nasty n.

In section 3.1, it has been mentioned that the quadratic congruence

$$x^2 \equiv y^2 \pmod{n}$$

is the point to start from. Having found such a congruence, where $x \neq y$, it can be expected that in every 2nd case it is

$$1 < gcd(n, x - y) < n,$$

i.e. $gcd(n, x - y)$ is a non-trivial factor of n.

5.1 Dixon's Idea

In 1981, John Dixon suggested a slightly different approach, namely to select random integers s and then compute

$$f(s) := s \times s \bmod n.$$

A *factor base* – a set $\mathcal{B}$ of small prime factors – it set up and it it is looked for all integers such that $f(s)$ can be completely factored over the factor base, i.e. all prime factors of $f(s)$ are contained in $\mathcal{B}$. The basic idea behind Dixon's algorithm is to then select certain tuples $(s, f(s))$ in such a way that every prime that is a member of the factor base is used an even number of times. If it is possible to reach that goal, then a congruence of the type $x^2 \equiv y^2 \pmod{n}$ is found. And that is all we want.

Example. Let $n := 187$ be an RSA-like modulus. Assume that three congruences have already been found:

$$\begin{aligned} 23^2 &\equiv 5 \times 31 \pmod{n}, \\ 24^2 &\equiv 3 \times 5 \pmod{n}, \\ 29^2 &\equiv 3 \times 31 \pmod{n}. \end{aligned}$$

Then it is known that

$$\begin{aligned} (23 \times 24 \times 29)^2 &\equiv (3 \times 5 \times 31)^2 \pmod{n}, \\ 113^2 &\equiv 91^2 \pmod{n} \end{aligned}$$

and the computation of

$$gcd(113 - 91, 187) = 11,$$

reveals the first prime factor of n.

Like above, $\mathcal{B} := \{p_1, p_2, ..., p_B\}$ shall be the factor base. Let C be a number slightly larger than B, e.g. $C := B + 10$. Suppose C congruences whose right side can be completely factored over the factor base have already been found. It is now possible to build up a matrix using the exponents of the integer's prime factors as its components. Gaussian elimination with the identical matrix as the right side will then show which congruences have to be combined in order to get the desired form. In that process, the identity is being transformed in such a way that is is only necessary to multiply the congruences which have got a 1 in the corresponding column of the right side of the matrix system in order to get a perfect square.

Example. Let once again $n := 187$ be the modulus. The factor base is $\mathcal{B} := \{2, 3, 5, 7, 11, 17, 31\}$, $B := 7$ and $C := 8$. The congruences

$$\begin{aligned}
20^2 &\equiv 2 \times 17 \pmod{n}, \\
22^2 &\equiv 2 \times 5 \times 11 \pmod{n}, \\
23^2 &\equiv 5 \times 31 \pmod{n}, \\
24^2 &\equiv 3 \times 5 \pmod{n}, \\
27^2 &\equiv 2^3 \times 3 \times 7 \pmod{n}, \\
28^2 &\equiv 2^2 \times 3^2 \pmod{n}, \\
29^2 &\equiv 3 \times 31 \pmod{n}, \\
33^2 &\equiv 2 \times 7 \times 11 \pmod{n}.
\end{aligned}$$

have been found. This allows to construct the following matrix system:

2	3	5	7	11	17	31		20	22	23	24	27	28	29	33
1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0
3	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0
2	2	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1

Since the actual exponents are not interesting at all as long as the sum of them is even, everything may be reduced modulo 2. The resulting system is:

2	3	5	7	11	17	31		20	22	23	24	27	28	29	33
1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1

Gaussian Elimination transforms that to

2	3	5	7	11	17	31		20	22	23	24	27	28	29	33
1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0
0	0	0	1	0	1	1	0	1	0	0	0	0	0	1	0
0	0	0	0	1	1	1	0	1	1	0	1	1	0	1	0
0	0	0	0	0	1	0	0	1	1	0	1	1	0	0	1
0	0	0	0	0	0	0	0	0	0	□	□	0	0	□	0
0	0	0	0	0	0	0	0	0	0	0	0	0	□	0	0

We are only interested in those rows that only contain 0's, i.e. prime powers whose exponents are $\equiv 0 \pmod{2}$. What we see is that the combination of congruences #3, #4 and #7 as well as congruence #6 meet that criterion, i.e. that $23^2 \times 24^2 \times 29^2$ and 28^2 are perfect squares. Of course, it is nothing new that $2^2 \times 3^2$ only contains even prime powers. But nevertheless, Gauss gives us a second linear combination that

works, too. We can now compute $gcd(113-91, 187) = 11$ and $gcd(28-6, 187) = 11$.

This method always works if $C > B$ because then the number of rows in our matrix gets bigger than the number of lines. In order to keep a quadratic matrix, it is filled up with a column full of 0's. That makes columns (and therefore the rows) linear dependend, i.e. there is at least one combination so that Gauss will generate a row full of 0's.

One final question still remains: The linear dependence desired could be achieved by simply setting $C := B + 1$. Why is it good to take a bigger C? The answer is that there is no guarantee that that the combination works. In about every 2nd case, we have got that

$$x \equiv \pm y \pmod{n} \;\Rightarrow\; gcd(x - y, n) = n.$$

This does not help at all, because everybody knows that $n|n$, and we are interested in finding non-trivial divisors. But if $C := B + 10$ for example, the probability that there is one combination that works is

$$P := 1 - \frac{1}{2^{10}} \;\approx\; 0.999$$

and that should be sufficient for most purposes. However, everybody has to find out him-/herself what value is best for his/her needs. $P = 0.999$ is quite a satisfying probability but imagine your computer having done some days or weeks of computation and then stopping and telling you that it has not been able to find a factor – just because you have chosen C 1 or 2 too small. If you build up a (1000×1000)-matrix – which indeed is one of the smaller ones when trying to factor really large integers – you do not need to be afraid of the additional computation caused by C being $B + 20$ instead of $B + 10$.

5.2 Pomerance's Improvements

In 1981, Carl Pomerance suggested incorporating a sieving algorithm – similar to the Sieve of Eratosthenes, which has been presented in Section 3 (Algorithm 3.1) – in the procedure described above. The sieving can be performed extremely fast and allows Trial Division on thousands of numbers simultaneously without really dividing them. Fortunately, the incorporation of a sieving mechanism is not the only improvement he made.

Improvement #1: Raising the probability of factorization over the factor base

Taking random integers x for creating the congruences $x^2 \equiv y^2$ causes some trouble. Assuming uniform distribution, it can be conclude that about $\frac{9}{10}$ of the y^2's are between $\frac{n}{10}$ and n, so that they are of a length approximately $|n| - 1$. Since n cannot be decomposed by Trial Division (yes, that's the reason why we are using the Quadratic Sieve!) we can't in fact believe that this is any different with those congruences. In clear words: Trial Division will get us nowhere. So, Pomerance had the idea that, instead of using random numbers, it could be wise to define a function

$$f(r) := r \times r - n$$

and then construct the congruences in the manner

$$r^2 \equiv f(r) \pmod{n}.$$

Starting with $r_0 := \lfloor\sqrt{n}\rfloor$, the numbers $f(r)$ to be factored over the factor base are relatively close to $\sqrt{n}$. Having r running from r_0 to $r_0 + 10^5$, for example, means that $\sqrt{n} \approx lowerbound \leq f(r) \leq upperbound \approx \sqrt{n} \times 10^5$, i.e. the Trial Division is run on numbera of about half the length of n, which alone is already a great improvement. The $f(r)$'s probability of being factorable can even be increased a little by accepting r's $< r_0$, yielding negative $f(r)$'s, and adding -1 as a (pseudo-)prime factor to the factor base.

Improvement #2: Reducing the size of the factor base

Remember that the point of interest is a set of integers $f(r)$ that are divisible by any members of the factor base. If an integer p divides

$$f(r) = r \times r - n,$$

that essentially means that

$$n \equiv r^2 \pmod{p},$$

i.e. n is a quadratic residue modulo p, thus the Legendre symbol (n/p) is $+1$ ([BRE1989, p. 88]). This is an important discovery, since it shows that only primes satisfying $(n/p) = +1$, which is about every second prime, have to be included in the factor base. [BRE1989, p. 97,98] also shows an efficient way to compute the Legendre symbol. Efficiency is, however, not that important, because the computation has only to be done for very few (e.g. 10,000) primes.

Improvement #3: The Sieve

If n is a quadratic residue modulo p, then

$$n \equiv s^2 \pmod{p} \quad \text{or} \quad n \equiv (-s)^2 \pmod{p}.$$

This means that if

$$r \equiv s \pmod{p} \quad \text{or} \quad r \equiv -s \pmod{p},$$

then p divides $f(r)$, which allows to sieve over a huge amount of congruences, because we know that not only

$$p \mid f(r) \quad \text{and} \quad p \mid f(-r),$$

but also that

$$p \mid f(r + kp) \quad \text{and} \quad p \mid f(-r + kp) \quad \forall\ k \in \mathcal{Z}.$$

Applying the division function only to congruences

$$r^2 \equiv f(r) \pmod{n}$$

satisfying the equivalence

$$r \equiv t \ (\text{or} - t) \pmod{p}$$

guarantees that only numbers will be divided that are divisible. There are no unsuccessful trial divisions any more! This is a great improvement. Finding $f(r)$'s decomposable over the base now means taking a set of, say 100,000, numbers r, performing the sieving algorithm and then taking every congruences out of the set that can be completely factored. If the number of them not sufficient yet, it is only necessary to take another 100,000 integers and sieve again.

Although there no more unsuccessful trial divisios, there are certainly wasted ones: Imagine an integer $f(r)$, about 30 digits long, being factored. Assume it is

divisible by some primes, e.g. k distinct primes, contained in the factor base, but not entirely decomposable. Then the algorithm in its current form would perform k divisions on that number, which are completely wasted, since that number cannot be used for constructing a perfect square because of the lack of a complete decomposition. The issue of successful, but wasted, divisions is addressed by the following

Improvement #4: Dividing an integer without performing any division

Suppose a number m can be completely factored over the base, i.e. that

$$\exists\ \vec{p} := (p_1, p_2, ..., p_k) : \qquad n = \prod_{i=1}^{k} p_i$$

$$\Rightarrow \quad log(n) = \sum_{i=1}^{k} log(p_i).$$

So, it is not even necessary to store the values of the $f(r)$'s, but only their logarithms. Instead of doing the division on every $f(r)$, where $r \equiv t\ (or - t) \pmod{p}$, it is only necessary to subtract off the logarithm of the respective prime p_i. If the remaining value, after performing that operation for all primes contained in the factor base, is sufficiently close to zero, the number in question is completely decomposable. Trial division can then be applied to that number in order to compute its complete decomposition. [BRE1989] suggests using single precision floating point arithmetic. However, I have experienced serious trouble even when dealing with relatively small integers (about 40 digits). Thus, it is strongly recommended to use double precision arithmetic in order to avoid severe rounding errors that keep you from enjoying the full power of the Quadratic Sieve algorithm.

Of course, not only the actual primes p may divide $f(r)$, but also any prime power p^a. Thus, it is necessary to also solve congruences of the type

$$x^2 \equiv n \pmod{p^a}.$$

This is bad, because it makes the algorithm sieve over small primes like 2 and 3 most of the time. Robert Silverman (cf. [BRE1989, p. 105]) suggested to ignore these higher powers and instead to be a little more generous when interpreting the remaining value after subtraction of the primes' logarithms. Doing so, it is no longer certain that an integer whose remaining logarithm value is small enough can be completely factored, but it is *likely*. This means that again a few wasted trial divisions are done during the final Trial Division phase, in which all numbers that are likely to be completely decomposable are examined, but that drawback is more than compensated by the speedup achieved by ignoring the higher powers.

5.3 Some subtle Refinements

Construction of the factor base

One of the most important decisions is the size of the factor base. While a smaller factor base means that Gauss runs faster (since the resulting matrix system is smaller) and that there are less unsuccessful divisions, a bigger factor base decreases the number of integers to be sieved necessary to construct a perfect square.
The actual size is tightly bound to the architectural specifications of the hardware used, but the following table can be used to determine approximations

a	$P(p < n)$
2	3.07×10^{-1}
3	4.86×10^{-2}
4	4.91×10^{-3}
5	3.55×10^{-4}
6	1.96×10^{-5}
7	8.75×10^{-7}
8	3.23×10^{-8}
9	1.02×10^{-9}

Table 5.1: Probability that the largest prime divisor of n is $p < n^{\frac{1}{a}}$. ([BRE1989, p. 106])

Solving quadratic congruences

In order to apply the sieve to all congruences built up, it is necessary to compute the solution to the congruence

$$t^2 \equiv n \pmod{p}$$

for each prime p. For small primes p, this can be done using some kind of brute force attack, i.e. trying all integers $0 \leq i < p$. That method quickly becomes inefficient as the primes are growing. So, faster methods are needed:

- if $p = 4k + 3$, then
$$t \equiv n^{k+1} \pmod{p}.$$
- if $p = 8k + 5$ and $n^{2k+1} \equiv +1 \pmod{p}$, then
$$t \equiv n^{k+1} \pmod{p}.$$
- if $p = 8k + 5$ and $n^{2k+1} \equiv -1 \pmod{p}$, then
$$t \equiv (4n)^{k+1} \times (\frac{p+1}{2}) \pmod{p}.$$

A formal proof as well as instructions on how to deal with the remaining case $p = 8k + 1$ is given in [BRE1989, pp. 107,108].

The sieving process

After building the factor base and calculating the $f(r)$'s for some 10,000 r's, the sieving process is started. Since higher prime powers are ignored by the sieve, it is no longer necessary (nor possible, in most cases) to have $log[r] = 0$ after subtracting all primes' logarithms. Hence, it is possible to add another probabilistic element: Rather than computing $log(f(r))$ for each r and then subtracting the primes' logs, $log[r]$ is initialized as zero and then the logarithms of the primes dividing $f(r)$ are begin added subsequently.

Silverman (cf. [BRE1989]) suggested that an integer should be considered as *possible* being able to factor completely if

$$log[r] > \frac{log(n)}{2} + log(m) - 2 \times log(p_{max}),$$

where m is the number of integers to be sieved over and p_{max} is the largest prime in the factor base.

5.4 Large Primes and Multiple Polynomials

Many suggestions have been made on how to improve the Quadratic Sieve. Since the actual sieving is the part of the algorithm that consumes the most time, the improvements concern the sieving. Two of the most sensible and successful improvements presented are the incorporation of large primes in the factor base and the use of multiple polynomials for constructing the congruences.

Incorporating large primes

Imagine the sieve having found an integer y_1 that can be factored partially and whose remaining quotient is

$$q_1 < p_{max}^2.$$

Obviously, q is prime. If a second integer y_2 can be decomposed to

$$q_2 \times \prod p_i \ , \quad q_2 < p_{max}^2,$$

where the p_i's are some elements of the factor base, q^2 is a perfect square and the product $y_0 \times y_1$ of the two integers can be added to the list of completely factored integers.

Note that it is neither necessary nor recommended to explicitly look for those large primes q, since that would imply a larger factor base, but if two such integers have been found, they are taken. Collision can be looked for using binary trees or hashes.

Multiple polynomials

Peter Montgomery (cf. [BRE1989, p. 116]) suggested using polynomials different to

$$f(x) = x \times x - n,$$

which has been used in the original Quadratic Sieve. The problem with this polynomial is that the numbers to be factored tend to be quite large (about $\sqrt{n} \times m$, where m is the number of integers to be sieved over). So, he used different polynomials that guarantee a lower upper bound of $f(x)$. Thus, more numbers can be completely decomposed. Another effect of its idea is that sieving can easily be parallelized: Take 100 computers and assign a different polynomial to each of them. In fact, this is the way modern factorization specialists work.

5.5 QS Summarized

The Quadratic Sieve consists of five main steps to be performed:

- Build up a factor base, so that every integer p contained in that base is prime and the Legendre symbol (n/p) is $+1$.
- Solve the congruence
$$t^2 \equiv n \pmod{p}$$
for every prime p in the factor base.
- Perform the sieving in order to find enough $f(r)$'s that can be completely factored over the factor base.

- Run Gauss' algorithm to find products of the $f(r)$'s that are a perfect squares, i.e. linear combinations of the rows, in which every prime exponent is even.
- For the resulting perfect squares, check if

$$1 < gcd(n, x - y) < n.$$

  If so, a non-trivial divisor of n has been found.

One thing uncovered yet is the estimated running time of the Quadratic Sieve: [STI1995, p. 155] states that the asymptotic running time necessary to find a divisor of n is

$$\mathcal{O}\ (e^{\sqrt{ln(n) \times ln(ln(n))}}).$$

6 Continued Fractions (CFRAC)

The major problem of all algorithms using the approach

$$x^2 \equiv y^2 \pmod{n}$$

is the necessity of running Trial Division on many numbers in order to construct a perfect square. In the case of QS, these numbers are approximately as large as $m \times \sqrt{n}$, where m is the number of integers sieved over. That upper bound can be reduced by means of Multiple Polynomials, but still remains $\frac{m}{\sqrt{2}} \times \sqrt{n}$. While the Quadratic Sieve accepts these large numbers and solves the problem by dividing many numbers simultaneously (using the sieving technique), the Brillhart-Morrison Continued Fractions Algorithm (CFRAC) addresses the issue in a different way.

Billhart and Morrison looked for a possibility to find integers r, so that $r^2 \bmod n$ is relatively small and thus likely to be factorable over the factor base. Their work, presented in 1975, is mainly based on research done by Lehmer, Powers in the the 1930s.

Suppose $m := \lfloor \sqrt{n} \rfloor$ and

$$\begin{aligned}
&a_1 := m,\ a_i := \lfloor (m + b_{i-1})/c_{i-1} \rfloor \quad \forall\ i \geq 2,\\
&b_0 := 0,\ b_1 := m,\ b_i := a_i \times c_{i-1} - b_{i-1} \quad \forall\ i \geq 2,\\
&c_0 := 1,\ c_1 := n - m \times m,\ c_i := c_{i-2} + a_i \times (b_{i-1} - b_i) \quad \forall\ i \geq 2,\\
&p_0 := 1,\ p_1 := m,\ p_i := p_{i-2} + a_i \times p_{i-1} \quad \forall\ i \geq 2.
\end{aligned}$$

From these definitions it can be concluded

$$p_i^2 \equiv (-1)^i \times c_i \pmod{n} \quad \text{and} \quad c_i < 2\sqrt{n} \quad \forall\ i\ \epsilon\ N.$$

A formal proof of this is given by [BRE1989, pp. 148-155].

Setting $r_i := p_i$ (and thus $f(r_i) := c_i$) allows to construct congruences with a lower upper bound than are more likely to be completely factorable over the factor base. However, note that although the upper bound $2\sqrt{n}$ is an impressive improvement compared to the $m \times \sqrt{n}$ of the Quadratic Sieve, it is still not good enough. The amazing speedup delivered by the sieving technique more than compensates for the slightly larger integers created. Therefore, CFRAC is not in use any more, because QS is always faster.

7 Benchmarks

Note: All benchmarking was done on an Intel Celeron machine (433 MHz), using the J2RE 1.3.0 by IBM. The Quadratic Sieve and Continued Fraction algorithms benchmarked are the basic versions, including neither the Large Primes nor the Multiple Polynomials refinement.

Factorization of small Integers ($< 10^{16}$)

n	$\|n\|$	Trial D.[1]	Trial D.[2]	Fermat	Poll. p-1	Pollard ρ
161655583	9	0.17 s	0.11 s	0.32 s	0.07 s	0.01 s
2284761119	10	0.49 s	0.36 s	0.48 s	1.91 s	0.01 s
54035059889	11	0.56 s	0.65 s	3.04 s	0.07 s	0.19 s
128851260341	12	1.60 s	0.93 s	1.09 s	1.50 s	0.29 s
4863543324427	13	5.24 s	1.24 s	16.96 s	2.77 s	0.50 s
24922017588527	14	10.57 s	3.92 s	44.32 s	0.30 s	0.46 s
323392936942541	15	49.98 s	6.27 s	100.33 s	6.41 s	0.72 s
6393484731306767	16	332.21 s	35.92 s	127.61 s	1.86 s	0.79 s

[1] naive, trying all odd integers [2] using the Sieve of Eratosthenes

Factorization of medium sized Integers ($< 10^{27}$)

n	$\|n\|$	Pollard p-1	Pollard ρ	QS	CFRAC
93839607998515309159	20	1.80 s	1.97 s	1.83 s	3.77 s
248250100221579530297	21	1.35 s	9.49 s	2.18 s	5.66 s
2396454047783835285091	22	13.47 s	10.05 s	2.30 s	5.49 s
22348931488950209689159	23		11.12 s	2.68 s	10.23 s
3058944677982788614806 79	24		18.77 s	2.27 s	8.19 s
6497555360541462872605837	25		72.13 s	3.13 s	23.41 s
1037351986610436797075502 7	26		26.32 s	3.52 s	40.93 s
2716911028663786697382235 67	27			5.15 s	68.94 s

The Quadratic Sieve and Continued Fraction algorithms were run using the optimum factor base size.

Factorization of large Integers ($> 10^{30}$)

$\|n\|$	QS ($\|\mathcal{B}\| = 500$)	QS ($\|\mathcal{B}\| = 1000$)	QS ($\|\mathcal{B}\| = 2000$)	CFRAC ($\|\mathcal{B}\| = 1000$)
31	6.3 s	7.7 s	22.5 s	272.9 s
34	28.2 s	15.8 s	27.0 s	899.2 s
37	350.3 s	92.7 s	61.5 s	3148.3 s
40		858.7 s	264.9 s	
43		1678.6 s	467.4 s	
46			995.3 s*	
49			2138.5 s*	

* $|\mathcal{B}| = 4000$ instead of 2000

References

[BRE1989] D. M. Bressoud. Factorization and Primality Testing. Springer-Verlag New York, 1989.

[BUN1998] P. Bundschuh. Einführung in die Zahlentheorie. Springer-Verlag Berlin, 1998.

[BUC2001] J. Buchmann. Einführung in die Kryptographie. Springer-Verlag Berlin, 2001.

[GAR1998] T. Garefalakis. Primality Testing, Integer Factorization and Discrete Logarithms. Toronto, 1998.

[MOV1996] A. Menezes, P. van Oorschot, S. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996. (www.cacr.math.uwaterloo.ca/hac)

[RAB1979] M. O. Rabin. Digitized signatures and public-key functions as intractible as factorization. *MIT Laboratory for Computer Science Technical Report*, LCS/TR-212, 1979.

[RSA1978] R. L. Rivest, A. Shamir, L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21 (1978), 120-126.

[SCH1996] B. Schneier. Applied Cryptography / Angewandte Kryptographie. Addison-Wesley, 1996.

[SHO1994] P. W. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. *Proceedings, 35th Annual Symposium on Foundation of Computer Science*, 1994, 124-134.

[STI1995] D. R. Stinson. Cryptography: theory and practice. CRC Press, 1995.