Integer Factorization

Diana Gren
Jonas Carlsson

DD2440 Advanced Algorithms
Stefan Nilsson

Contents

Chapter 1

Background

Factorize integers into prime numbers is considered a difficult problem. There is algorithms that solve the problem, but they can in some cases be very time consuming. An example is a product of two large prime numbers, which is very difficlt to factorize. This is the key in RSA cryptography, which completely relies on the fact that factoring products of large prime numbers is incredibly difficult.

1.1 Algorithms

As mentioned above, there are some algorithms that solve the problem, of which two of them will be discussed in this section.

1.1.1 Trial Division

Method

The most straight forward way of solving this problem is of course to divide a given number N with each prime number up to $\sqrt{N}$. If we find a prime x such that

$$x \equiv 0 \pmod{b} \tag{1.1}$$

we have found two factors. The first factor being x and the second being $\frac{N}{x}$.

Performance

Trial division is not an ideal approach as it is a heavy algortihm. It can, however, find a small factor fast if such a factor exists. Since there is a 50% chance that a large N is even, and 33% that it has a factor 3 and so on, it can be a good idea to check for small prime numbers as factors.

1.1.2 Fermat's Factorization Method

Method

A simple way of improving the naive method is to use the fact that every odd number number N can be represented as a difference of squares (proof in section A.1.

$$N = a^2 - b^2 \tag{1.2}$$

If we know a and b, N can easily be factored into two factors according to mathematical laws

$$N = a^2 - b^2 = (a + b)(a - b) \tag{1.3}$$

To find such integers, one starts with an $x_0 = \lceil \sqrt{N} \rceil$. Check if

$$\Delta x_0 = x_0^2 - N \tag{1.4}$$

is a square number. If it is not, try next; $x_1 = x_0 + 1$

$$\Delta x_1 = x_1^2 - N \tag{1.5}$$

etc. Do the same step, increasing x by one, until a square number Δx is found.

Performance

Fermat's factorization method works well when the resulting factors ara of similar size. If that is not the case, this method is not optimal. However, the method sets base for more advance and better performing algorithms such as the *quadratic sieve* and *general number field sieve*

1.1.3 Pollard Rho

The Pollard Rho algorithm was invented by John Pollard in 1975, and its speciality is to factor composite numbers with small factors.

Method

Performance

Chapter 2

Approach

We decided to do the programming in C++. We both had limited knowledge in the language and wanted to learn more, as well as handling I/O for Kattis felt like a much easier task in C++ than in Java.

Our approach for solving the problem was to implement the Pollard Rho algorithm. We did some research on the Quadratic Sieve, but ended up with Pollard Rho mostly because it is an easy algorithm, and at the same time it can perform quite well. We tested it towards Kattis, and tried to optimize it in different ways and in this chapter, we will discuss more into detail what was tested.

2.1 Pollard Rho

2.1.1 Implementation

We quickly realized that we needed the program to stop looking for factors if it was taking too long, so we introduced a cut off limit. The cut off limit is a limit for the maximum number of iterations the algorithm is allowed to do, and gives the answer f̈ailïf it did not succeed in fatorizing the given number.

The scores could be improved just by increasing the cut off limit so that the program would run for as close to 15 seconds as possible, since it obviously had time to evaluate more numbers.

```
f(z, N)
        return z*z % N
function pollard(N)
        x := random(N)
        y := x
        prod := 1
        for i := 1 to infinity do
                x = f(x, N)
                y = f(f(y, N), N)

                if (x - y = 0) then
                        prod = prod *(x-y) % N
                        d := gcd(N, prod)

                if d > 1 and d < N then
                        return d

                if i >= CUT_OFF_LIMIT then
                        return 0
        end for
end function
```

2.1.2 Optimizations

We recognized that since there is a 50/50 chance that the given number is even, it could be a good idea to check that before running the algorithm, and thereby eliminate calculation time. So we started off by introducing a check for even numbers, and just return true if the condition was satisfied.

After reading more on Pollard Rho, we found that finding the greatest common divisor is quite expensive to do, hence we do not want to do that so often. This made us multiply the value a certain number of times before entering the gcd, hoping to improve the algorithm.

2.2 Pollard Rho with Brent

Pollard-Rho is a great algorithm but it suffers from one flaw. It can't handle a cyclic behaviour. With a cyclic behaviour i mean the function f produces repeated results. We can take an example from our own pollard-rho function. When we tried to factor 25 we got this sequence:

$$a_1 = 1, a_2 = 22, a_3 = 16, a_4 = 2, a_5 = 19, a_6 = 7, a_7 = 2, a_8 = 19...$$

Where a is the value to be investigated with GCD. This sequence will go nowhere and we will have to abort this run.

2.2.1 Implementation

Cycle finding can be implemented with either Floyds ẗortoise and hare algorithmör with Brents improvement. We choose to implement Brents but we shall start by discussing Floyds. Floyds algorithm builds upon the idea that for integers $i >= \mu$ and $k >= 0, x_i = x_{i+k\lambda}$, where λ is the length of the loop. Especially when $i = k\lambda >= \mu$, it follows that $x_i = x_{2i}$. Thanks to this we can find a length v by letting one value take one step and the other take two steps. This value v is divisable with λ. Which is what we are searching for.

```
f(z, N)
        return z*z % N
function pollard_breant(N)
        y:=x0, r:=1, q:=1;
        while (G = 1)
                x=y;
                for i:=1 to r do y:=f(y,N), k:= 0
                        while (k>=r) or (G>1)
                                ys := y
                                for i:=0 to min(m, r-k) do
                                        y:= f(y); q = (q * |x-y|) mod N
                                end
                        end
                        G := GCD(q,N); k:=k+m
                end
                r=r*2
        end
        if G = N then
                while g = 1 do
                        ys = f(ys,N); G= GCD(|x-y|, N)
                end
        end
        if G = n then success else fail
end
```

2.2.2 Problems

Finding information on how to implement pollard-rho with brent proved hard as the general information we found was for the cycle finding algorithm. We did not find any information to the combined version of pollard-brent at first. But later we found the the paper that brent himself published. Which resulted in the current iteration of our code.

Chapter 3

Results

Cut off limit	Score	Time [s]
130	12	1.2
1000	15	2.4
10000	32	12.1
12500	39	14.20

Chapter 4

Conclusion

Appendix A

Proofs

A.1 Fermat Difference of Squares

Theorem: An odd integer N can be represented as a difference of squares $N = a^2 - b^2$.

Proof:

Let $N = m_1 m_2$, with $m_1 \leq m_2$. Since we know that N is odd, m_1 and m_2 must both be odd as well.

Let $a = \frac{1}{2}(m_2 + m_1)$ and $b = \frac{1}{2}(m_2 - m_1)$. Since both m_1 and m_2 are odd, both a and b will be integers.

This gives us $m_1 = a - b$ and $m_2 = a + b$, hence $N = m_1 m_2 = (a - b)(a + b) = a^2 - b^2$.

Bibliography

[1] WolframMathWorld. *Fermat's Factorization Method*
http://mathworld.wolfram.com/FermatsFactorizationMethod.html
Retreived: Oct 2012

[2] Department of Mathematics, Kansas State University. *Factoring Methods*
http://www.math.ksu.edu/math511/notes/925.html
Retreived: Oct 2012

[3] Math & Statistics Solutions. *Fermat's Factoring Technique*
http://mathsolutions.50webs.com/fermat.html
Retreived: Oct 2012