AVALG13

Integer Factorization

November 9, 2013

Carl Eriksson **carerik@kth.se**
Viktor Mattsson **vikmat@kth.se**

ABSTRACT

Integer factorization is the art of decomposing non-prime numbers as multiplication of prime numbers. It is a very taxing problem for computers to solve and therefore several algorithms has been invented. This report will mainly focus on the algorithms invented by the british mathematician John Pollard such as Pollard's rho and Pollard's P-1. In conclusion we discovered that using Pollard's rho with Brent"s cycle finding algorithm was an efficient way to factorize numbers though the best performance was achieved by combining several algorithms.

SAMMANFATTNING

Heltals faktorisering är problemet att dela icke-primtal till multiplicationer av primtal. Det är ett mycket kostsamt problem för datorer att lösa och därför har flera algoritmer uppfunnits. Denna rapport kommer främst att fokusera på algoritmerna den brittiske matematrikern John Pollard uppfunnit såsom Pollard's rho och Pollard's P-1.Sammanfattningsvis upptäckte vi att Pollard's rho med Brent's cycle finding algorithm gav ett effektivt sätt att faktorisera tal men det bästa resultatet åstadskoms genom att kombinera flera algoritmer.

Contents

1 Introduction

1.1 Statement of collaboration

1.2 Problem statement

Prime numbers are a quite interesting set of numbers. They share the special trait that they are only divisible by one or themselves, unlike non-primes. However, every non-prime number can be written as a series of prime number which is useful when computing data with large integers as it allow dividing a big problem into several smaller ones. Finding this series of primes or more commonly, finding the factors of these large integers is however no simple matter and efficient algorithms are needed.

1.3 Definitions

- N - Number to be factorized
- GCD - Greatest common divisor
- Kattis - Code "judge" used by KTH to test code

2 Algorithms

2.1 Used

2.1.1 Trial division

Trial division is one of the most inefficient algorithms for factorization of large integers. It is however one of the most efficient algorithms for factoring small integers and it also the easiest algorithm to perceive. The trial division algorithm only have a few steps. It begins with finding a set of prime-numbers using some algorithm such as the sieve of eratosthenes then iteratively check if N divided by any of these primes grants no remainder. If it does not grant any remainder the prime used in the division is a factor. The algorithm will continue checking this until N becomes one. A simple optimization of trial division is to never check primes greater than the square root of N. Considering that if $N = a * b$ and if a is greater than $\sqrt{N}$ then b must be smaller than $\sqrt{N}$. So for every factor greater than $\sqrt{N}$ there has to be one factor smaller than $\sqrt{N}$ thus if we check for factors up to $\sqrt{N}$ we will find a factor unless N is already prime.

2.1.1.1 Pseudocode:

Algorithm 1 Trial division

```
function TRAIL_DIVISION(N)
    if N = 1 then
        return
    end if
    list_of_primes = sieve_of_eratosthenes(√N)
    prime_factors = {}
    for all prime in primes do
        while n mod prime = 0 do
            prime_factors.add(prime)
            n = n/p
        end while
    end for
    if n not equal to 1 then
        return prime_factors.add(n)
    end if
    return prime_factors
end function
```

2.1.2 Pollard's rho

Pollard's rho is a factorization algorithm invented by John Pollard [1] It is also known as Pollard's Monte Carlo factorization algorithm due to it random properties and does not always manage to return a result. However, when it does return a result it always return a correct one. The main idea of Pollard's rho is that if you pick a random number x, then pick another random number y where both are between zero and N, you can check whether $(x - y)$ and N has a GCD which is not equal to one. If it does it is all fine but if it does not, you would have to find another random number z and check whether $(z - x)$ or $(z - y)$ has a GCD with N that is not equal to one. If it still does not grant a GCD which is not equal to one you will have to keep introducing random numbers and check if they work. As GCD is a very expensive operation picking numbers completely random may not be profitable. Therefore a typical choice for picking the numbers is using the formula $x^2 + a$ where a is a constant. This a can be chosen at will and certain a's are better than others for specific N's. Important to notice is that N can obviously not have any factors greater than itself thus the algorithm should be run in modulo N number space to repeat the process incase we reach numbers greater than N.

The algorithm will appear as cycling to the answers and whenever a cycle has been detected the algorithm will have found a solution. Note that a solution can in some cases include N even if N is not a prime. This is a result that the algorithm failed to find a factor. Pollard's rho uses Floyd's cycle finding algorithm to discover these cycles.

2.1.2.1 Floyd's cycle finding algorithm

Floyd's cycle finding algorithm is used to find cycles of calls in a program in order to prevent the program from running infinitely. Consider a small program that make a sequence of function calls such as this one

---

$x = 1$
while $x \neq 5$ **do**
    $x = f(x)$
end while

If the function f in this program would return numbers such as the sequence

$$f(1) = 2$$
$$f(2) = 3$$
$$f(3) = 1$$

it would result in an endless loop where the program keep running in cycles. The solution to this is to use two pointers which move at different speeds. This is commonly referred as a tortoise and hare algorithm. As two pointers move at different speeds, one will move much further ahead than the other at a certain time. However, if there is a cycle both of them will sooner or later get stuck in it and therefore you could easily check if a cycle exists by checking if the two pointers is located at the same spot.

2.1.2.2 Pseudocode:

Algorithm 2 Pollard rho

```
function POLLARD(N)
    if N mod 2 = 0 then
        return 2
    end if
    x = 2
    c = 1
    y = x
    g = 1
    while g = 1 do
        x = f(x)
        y = f(f(y)
        g = gcd(x − y, N)
    end while
    if g = N then
        return fail
    end if
    return g
end function
function F(N)
    return (N ∗ N + c)%N
end function
```

2.1.3 Pollard Brent

The basic structure of the algorithm is the same in the Pollard's rho algorithm with the exception of using Brent's cycle finding algorithm [2] instead of Floyd's.

2.1.3.1 Brent's cycle finding Brent's cycle finding algorithm is uses the same structure as Floyd's algorithm with two pointers that that move is different speed. The difference of Brent's compared against Floyd's is that the fast moving pointer's speed is increasing with the power of two and the slow one does not move at all. Instead this slow pointer teleports to the location of the fast one after each iteration. Brent's research found this algorithm to be 24% faster than Floyd's [2].

2.1.3.2 Pseudocode:

Algorithm 3 Pollard Brent

```
function POLLARD_BRENT(N)
    if N mod 2 = 0 then
        return 2
    end if
    y = random(N − 1)
    c = random(N − 1)
    m = random(N − 1)
    g = 1
    r = 1
    q = 1
    while g = 1 do
        x = y
        for all i in 0 → r do
            y = ((y ∗ y)%N + c)%N
        end for
        k = 0
        while k < r AND g = 1 do
            ys = y
            for all i in 0 → min(m, r − k) do
                y = ((y ∗ y)%N + c)%N
                q = q ∗ (|x − y|)%N
            end for
            g = gcd(q, N)
            k = k + m
        end while
        r = r ∗ 2
    end while
    if g = N then
        return fail
    end if
    return g
end function
```

2.1.4 Pollard's P-1

Pollard's P-1 uses Fermat's little theorem to find factors. Fermat's little theorem state that if $i^p = i \bmod p$ where p is a prime number, p is a factor of i. While this is slightly inconvenient for finding factors this theorem can be divided by i to create a more useful version. If $i^{(p-1)} = 1 \bmod p$, then $i^{(p-1)} - 1$ is a factor of i. Using this idea Pollard's P-1 try to find the GCD

between a coprime constant and a prime from a pre calculated set of primes within the modulo number space of N in hope to find a factor.

2.1.4.1 Pseudocode:

Algorithm 4 Pollard p-1

```
function POLLARD_P-1(N)
    c = 2
    for all prime in list of primes do
        p = prime
        while p < last prime in list of primes do
            c = c^prime %_0 N
            p = p * prime
        end while
    end for
    g = gcd(c − 1, N)
    if 1 < g < n then
        return g
    else
        return fail
    end if
end function
```

2.1.5 Miller-Rabin primary test

Miller-Rabin primary test is a monte carlo algorithm with the time complexity of $\mathcal{O}(k * log(n)^2)$. The probability for the primary test is 2^{-k} where k is the number of iterations.

Since we used the primary test in gmp that uses Miller-Rabin primary test(after trial-division) we won't present any pseudocode for this algorithm.

2.2 Considered

2.2.1 Quadratic sieve

We considered to implement the quadratic sieve but due to the hardness of the implementation and the lack of good describing pseudo code we decided not to implement it.

2.2.2 Fermat factorization

The key concept of fermat factorization is that prime can be written in the form $N = x^2 - y^2$ which itself can be rewritten into $N = (x + y)(x - y)$. Thus search for an x that satisfy $x^2 - N = y^2$ equation. Then a non trivial prime factor is found [3].

After reading up on the algorithm and its performance we decided not to put time on implement it.

2.2.2.1 Pseudocode:

Algorithm 5 Fermat factorization

```
function FERMAT(N)
    a = ⌈√N⌉
    b2 = a * a − N
    while b2 is not square do
        a = a + 1
        b2 = a * a − N
    end while
    return a − √b2
end function
```

3 Result

The algorithm that gave the best result in kattis [4] by it self was Pollard Brent algorithm, with the score of 76 points. The best implementation overall was a combination of all the algorithms in the order Trial Division, Pollard Brent, Pollard's P-1 and Pollard's rho. This combination granted a score of 81 points on kattis ID 454156.

3.1 Trial division

Trial division were never use as a stand alone solver. It is known to be inefficient by itself and was therefore only used as a complement for finding small factors before running the other algorithms. Thus there is no specific result for trial division.

3.2 Pollard rho

Pollard rho granted 68 points in kattis.

3.3 Pollard Brent

The first version of pollard brent granted a humble 50 points in kattis. The second version granted 60 points. The third version granted 75 points. The fourth version granted 76 points and was the best overall.

3.4 Pollard's P-1

Pollard's P-1 was only use in combination with all the other algorithms and got a result of 81 points in kattis.

4 Discussion

Before we actually sat down and started to program we did research on the algorithms and deciding which ones we we're going to implement. We agreed on starting with the basic Pollard rho and thereafter continue improving it by rewriting it to Pollard Brent.

During our research we clearly understood that the algorithm that gave the best result was the quadratic sieve. We tried to find any good paper on how to implement the algorithm but we found mostly mathematical papers with fuzzy instructions. Hence we decided not to try to implement in and rather try to implement a good Pollard Brend and try to optimize it instead.

At the end we noticed that the algorithms gave good result on different type on N. Thus for the final version we combined all the algorithms we had implemented and spend a great deal of time to find breaking points for each algorithm though it resulted in an even better score.

4.1 Implementation details

4.1.1 Trial division

Trial division were used in both Pollard's rho and Pollard Brent algorithms for optimization purpose. From the beginning we made a list of small prime numbers to try out before running other algorithms. We decided that the twenty to forty first prime numbers would be enough to ensure that the majority of non-prime N's should get at least one factor from this without making it too costly.

4.1.2 Pollard's rho

For Pollard's rho we implemented it directly from its pseudo code and it work straight away. We did try alter parameters such as constants, skipping

the larger numbers and setting a limit for the iteration to make sure we do not spend too much time on a specific number. No other optimizations were done.

4.1.3 Pollard Brent

Pollard Brent is where most our work was done. It was also the first algorithm to be implemented as it were not too hard to write and were supposively superior to Pollard's rho. However we did have several setbacks with it.

On the first implementation we felt that it were not working very well. At least our institution were that something must be wrong. We did not follow the algorithm exactly at this point but followed some strange posts from the internet about how to make it more efficient such as using primes for constants and change these after a few iterations. It proved to be incorrect thus making us move on to a second version.

The second implementation was more closely based on what Pollard Brent really is. Removing the so called "optimizations" we read about such as using primes for constants and changing these over time. Instead we set them to one and also reduced the number of checks of how long time the program has been running. This in all did not grant much of a performance increase but changing the value m (used to count iterations before GCD) to the numbers bit length resulted in a much greater performance increase. At this time we also had a strange idea about changing the initial random value.

Using the fact that any non prime number should have a factor that is less than or equal to its square root we decided to try out changing the random from giving a number up to N to a number up to $\sqrt{N}$ for our starting-point in hope that we would end up finding this factor faster. This also granted us a quite big performance increase though we were unsure about if it really does work.

5 Conclusion

The final program that managed solve 81 of the 100 tests on Kattis, used a combination of all the algorithm we implemented. It seems like the general solution to integer factorization is to use different algorithms depending on N.

References

[1] *A monte carlo method for factorization*. Nov. 2013. URL: https://www.cs.cmu.edu/~avrim/451f11/lectures/lect1122_Pollard.pdf.

[2] *An improved monte carlo factorization algorithm*. Nov. 2013. URL: http://maths-people.anu.edu.au/~brent/pd/rpb051i.pdf.

[3] *Factoring large integers*. Nov. 2013. URL: http://www.ams.org/journals/mcom/1974-28-126/S0025-5718-1974-0340163-2/S0025-5718-1974-0340163-2.pdf.

[4] *Kattis*. Nov. 2013. URL: https://kth.kattis.scrool.se/.