AVALG13

Integer Factorization

November 8, 2013

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

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. 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 = random()
    c = random()
    y = c
    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 error
    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 instead of Floyd's.

2.1.3.1 Brent's cycle finding algorithm

TODO

2.1.4 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.

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

2.2.2.1 Pseudocode:

Algorithm 3 Fermat factorization

```
function FERMAT(N)
    a = ciel(√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
```