Humboldt University Berlin
Stastics
Location: Berlin
Winter term 2018/2019
Course: Numerical Introductory Seminar
Examiner: Dr. Petra Burdejova

Prime Factorization

Anna Friesen
533688
frisenan@student.hu-berlin.de
12th of February 2019

Contents

1. Introduction

To motivate the topic it will be shown, that prime factorization is very important in our days. In general data has to be encryted using cryptosystems. One example is the RSA cryptosystem. It was published in 1977 at the MIT by Ron **R**ivest, Adi **S**hamir and Leonard **A**delman. The cryptosystem ist named by the letters of the men's last names - RSA.

The algorithm is based on two pairs of keys. The first pair (N, E) ist used to encrypt the data. the second pair (N, D) is used to decrypt the data. N is the product of two large prime numbers p and q. In this case "large" means, that the number of decimal digits of the primes p and q should be larger than 232, i.e. $\log(i) > 232, i = p, q$.

The product N of p and q cannot be factored by now. That means, that it is computationally intractable for classical (non-quantum) computers. Using quantum computers Shor's algorithm could factor N. But since there are no quantum computers used , the RSA cryptosystem is still used to encrypt and ist still secure.

1.1. The RSA algorithm

Let OI be the original integer to be encrypted. If a string should be encrypted, the text has to be transfered into an integer. Here the number of each letter in the alphabet can be used.

1. choose two random primes p, q
   - $p \cdot q = N > OI$ and $p \neq q$ and $\mid p - q \mid$ not too small
2. choose E (E prime suffices all conditions)
   - $E \in \{n \in \mathbb{N} \mid 2 \nmid n\}$ and $E \nmid N$ and $E > (p-1) \cdot (q-1)$
3. choose D
   - $(E \cdot D) mod((p-1) \cdot (q-1)) = 1$

The two pairs of keys can be used in the following way.

$OI \xrightarrow[\text{encrypt}]{} CI : CI = OI^E mod(N)$

$CI \xrightarrow[\text{decrypt}]{} OI : OI = CI^D mod(N)$

2. Prime Factorization

2.1. Trial Division

The Trial Division ist the simplest and most simple-minded prime factorization algorithm. Assuming n is not prime, for $2 \leq i \leq \lfloor \sqrt{n} \rfloor$ it is tested whether i diveds n, i.e. $i \mid n$. It works quite well, because most coposite numbers have small prime factors. But it ist inefficient, if n has large prime factors. Hence it would be inefficient factoring N of the RSA algorithm.

In this case an **efficient** algorithm has a **polynomial** running time $f(\log_2(n))$, i.e. $f(\log_2(n)) \in O((\log_2(n))^k)$ mit $k \geq 0$.

Possible modifications of the Trial Division are the following.

The first modification would be to choose only odd numbers for i to test whether they divide n, i.e. $i \in \{k \in \mathbb{N} \mid 2 \nmid k\}$.

In the next modification there is a fixed bound B chosen to find i, i.e. $i \leq B$ (often $B = 10^6$).

The last modification is, to test only primes i that are smaller or equal to a bound r, i.e. $i \in \mathbb{P} : i \leq r$. The sieve of Eratosthenes returns a list of prime numbers that are smaller or equal to a fixed bound.

2.1.1. Sieve of Eratosthenes

The sieve of Eratosthenes is named by the greek mathematician Eratosthenes of Cyrene. he lived in the 3rd century BC. This mathematician invented the name *sieve* for a known algorithm. It is a deterministic algorithm, which creates a list of primes that are smaller than or eqal to a given integer $r \geq 2$. The running time of the algorithm is *exponentional*, i.e. $\log_2(r-1) \cdot (r-1)$.

The R code for the sieve of Eratosthemes can be found on git hub (https://gist.github.com/seankross/5946396) and it is listed in the Appendix.

2.2. Modern factorization algorithms

Factorization algorithms can be devided into two groups, The *special purpose algorithms* and the *general purpose algorithms*.

For the special purpose algorithms the efficiency depends on factors of the number being factored. One example for this type of factorization algorithm is the Pollard (p-1) factorization method. This factorization method is not dangerous to RSA, because RSA uses large primes to calculate the product N.

The efficiency of the general purpose algorithms depends only on number being factored. An example would be the general number field sieve.

2.3. Pollard (p-1) factorization method

The Pollard (p-1) factorization method is based on Fermat's little theorem

Theorem 1 (Fermat's little theorem) *Let p be prime ($p \in \mathbb{P}$) and $a \in \mathbb{Z}$ with $a < p$. Then it holds:*

$$p \in \mathbb{P} \Rightarrow a^p \equiv a\,(modp)$$

If additionally a and p have no common divisor, i.e. $\gcd(p,a) = 1$*, then it even holds:*

$$p \in \mathbb{P} \Rightarrow a^{p-1} \equiv 1\,(modp)$$

This theorem cannot be used as a primality test by itselfe, because the opposite direction $\Leftarrow$ does not hold.

The Pollard (p-1) factorization method was invented by John Pollard in 1974. It is a derivative of the Pollard rho method. The Pollard (p-1) factorization method is suitable for an integer n that is B-smooth for $p-1$.

Here a smooth (or friable) number is an integer which factors completely into small prime numbers. For example, a 7-smooth number is a number whose prime factors are all at most 7.

The Pollard (p-1) factorization method uses the fact that for any a and $\forall p \in \mathbb{P}$: $a^{p-1} \equiv 1\,(\mathrm{mod}\,n)$, i.e. $a^{p-1} - 1 \equiv 0\,(\mathrm{mod}\,n)$. The algorithm of Pollard's rho method is very easy.

1. Pick two random numbers: $x\,(\mathrm{mod}\,n)$ and $y\,(\mathrm{mod}\,n)$

2. If $x - y = 0 \,(\mathrm{mod}\, n)$ we found a factor $gcd(x - y, n)$, else go to step 1

The algorithm of the Pollard (p-1) factorization method is the following. It was designed from the outset to cope with the possibility of failure to return a result.

Choose a test cap B and call Pollard's p-1 algorithm with a positive integer.

1. Use the sieve of Eratosthenes to find primes $p < B$.
2. Choose an integer a to be coprime to n. If n is odd then one possible choice is $a = 2$. For even n, it could be cosen $a = \lfloor n \rfloor + 1$.
3. Find the exponent e such that $p^e \leq B$. Then for each $p < B$, compute $b = a^{p^e} \bmod n$ and see if $1 < \gcd(b - 1, n) < n$. If that's the case, return those results of the greatest common divisor function and exit. Now one prime factor of n is found.
4. If the GCD function consistently returned 1s, one could try a higher test cap B and try again from step 1.
5. If the GCD function consistently returned n itself, this could indicate that a is in fact not coprime to n, in which case one could try going back to step 2 to pick a different a.
6. Throw a failure exception.

2.3.1. The Pollard (p-1) factorization algorithm in R

First of all the package FRACTION is needed to call the faunction gcd() to calulate the greatest common divisor.

```
library("FRACTION")
#install.packages("FRACTION")
```

Then the function sieveOfEratosthenes() is written to gain a vector of primes that are smaller or equal to a fixed number. This function can be found on github, as described above. The only modification was the inclusion of the error text, that demands an input that is greater than 3, because the R code only works for such input numbers.

```
if(num < 4){
    return("Please enter a natural number that is larger than 3.")
    break
  }
```

The Pollard (p-1) function is declared. It demands an input n, which is the number that should be factored. Furthermore there is a default value for B implemented. This has to be 4, because later on sieveOfEratosthenes(B) is calculated and it is known, the the input has to be at least 4.

```
pollardPminusOne <- function(n, B = 4){
  if(n < 3){
    return("Please enter a value that is larger than 2.")
    break
  }
```

If n is odd, $a = 2$.

```
  else{
    if(n %% 2 != 0){
      a = 2
    }
```

If n can be divided by 2, $a = floor(sqrt(n)) + 1$.

```
    else{
      a = floor(sqrt(n))+1
    }
```

If B is a prime number, the sieve of Eratosthenes returns B as the last number in the returned vector. But there are only needed the prime numbers that are smaller than B. So if the vector includes B, the last value is deleted and the new vector is named *primes*.

```
    m = sieveOfEratosthenes(B)[length(sieveOfEratosthenes(B))]
    if( m == B){
      primes = sieveOfEratosthenes(B)[-length(sieveOfEratosthenes(B))]
    }else{
      primes = sieveOfEratosthenes(B)
    }
```

For every prime number that is smaller than B, there is calculated the exponent e such that $p^e < B$. These powers are stored in the *primes* vector.

```
    for(i in 1:length(primes)){
      p = primes[i]
      e = 1
      while(p^e < B){
        primes[i] = p^e
        e = e+1
        if(((p^e) < B) | ((p^e) == B)){
          primes[i] = p^e
        }
      }
    }
```

The two counter variables *counterOne* and *counterN* are set to be 0.

```
  counterOne = 0
  counterN = 0
```

For each power p^e it is calculated $a^{p^e} \bmod n$ and then the greatest common divisor $f = \gcd((a^{p^e} \bmod n) - 1, n)$.

If the greatest common devisor is greater than 1 and smaller than n, it is returned and the first prime factor of n is found. If the GCD is equal to 1, *counterOne* is counted up and if the GCD is equal to n, *counterN* is counted up. In all other cases the Pollard (p-1) method failed.

```
  for(i in 1:length(primes)){
    b = (a^primes[i]) %% n
    f = gcd((b-1),n)
    if((1 < f) & (f < n)){
      return(f)
      break
    }else if(f==1){
      counterOne = counterOne + 1
    }else if(f==n){
      counterN = counterN + 1
    }else{
      return("Failure")
      break
    }
  }
```

If the *counterOne* variable equals the length of the *primes* vector, it means that the GCD function consistently returned 1s. In this case the pollardPminusOne()

function is called with a B that is counted up one time, i.e. $B = B + 1$. The case that the GCD function consistently returned n is not implemented in this function. In the other cases n is a prime number and has no prime factors, that are smaller than n. So 1 or n ist returned.

```
    if(counterOne == length(primes)){
      pollardPminusOne(n, B = B+1)
    }else{
      return(f)
    }

  }
}
```

Next there are listed the function calls and the outputs of the function test.

```
> pollardPminusOne(1)
[1] "Please enter a value that is larger than 2."
> pollardPminusOne(2)
[1] "Please enter a value that is larger than 2."
> pollardPminusOne(3)
[1] 1
> pollardPminusOne(4)
[1] 2
> pollardPminusOne(5)
[1] 1
> pollardPminusOne(6)
[1] 2
> pollardPminusOne(7)
[1] 7
> pollardPminusOne(8)
[1] 2
> pollardPminusOne(9)
[1] 3
> pollardPminusOne(10)
[1] 5
> pollardPminusOne(12)
[1] 3
> pollardPminusOne(27)
```

```
[1] 3
> pollardPminusOne(91)
[1] 7
> pollardPminusOne(119)
[1] 7
> pollardPminusOne(437)
[1] 23
> pollardPminusOne(667)
[1] 23
> pollardPminusOne(899)
[1] 31
```

3. RSA-challenge

The RSA cryptosystem is still used to encryt data, so there exists a challenge that is called the RSA-challenge. the task of the challenge is to factor the product N of two large prime numbers p and q. In 2009 RSA-768 was factored over the span of two years. It is 768 bits and 232 decimal digits of size and it is the largest solved RSA-challenge so far. It was calculated with the general number field sieve.

RSA-768 = 334780716989568987860441698482126908177047949837
137685689124313889828837938780022876147116525317
43087737814467999489
x
367460436667995904282446337996279526322791581643
430876426760322838157396665112792333734171433968
10270092798736308917

4. Conclusion

As it was shown prime factorization is very important in our days, because it asures the security of the popular RSA cryptosystem. For now there are no factorization algorithms factor the RSA product N that is calulated using two large prime numbers. The RSA challenge motivates members to factor these RSA numbers. For small prime factors the trial division suits well, but for large prime factors a general purpose factorization algorithm should be used. The Pollard (p-1) factorization method is an example for a special purpose factorization algorithm and was implemented in R for this seminar. The R code can be found on git hub under https://github.com/annafriesen/pollard_s_p-1.

References

Lasse Rempe, Rebecca Waldecker
Primzahltests für Einsteiger
1. Auflage, Vieweg+Teubner Verlag, Wiesbaden

Martin Dietzfelbinger
Primality Testing in Polynomial Time - From Randomized Algorithms to "'PRIMES is in P"'
1.Auflage, Springer Verlag, Berlin Heidelberg

Richard P. Brent
Recent Progress and Prospects for Integer Factorisation Algorithms

Thorsten Kleinjung and Kazumaro Aoki and Jens Franke and Arjen Lenstra and Emmanuel Thomé and Joppe Bos and Pierrick Gaudry and Alexander Kruppa and Peter Montgomery and Dag Arne Osvik and Herman te Riele and Andrey Timofeev and Paul Zimmermann
Factorization of a 768-bit RSA modulus
Cryptology ePrint Archive, Report 2010/006
available on https://eprint.iacr.org/2010/006, 2018

Appendix

A. R code for the sieve of Eratosthenes

```
sieveOfEratosthenes <- function(num){
  if(num < 4){
    return("Please enter a natural number that is larger than 3.")
    break
  }
  values <- rep(TRUE, num)
  values[1] <- FALSE
  prev.prime <- 2
  for(i in prev.prime:sqrt(num)){
    values[seq.int(2 * prev.prime, num, prev.prime)] <- FALSE
    prev.prime <- prev.prime + min(which(values[(prev.prime + 1) : num]))
  }
  return(which(values))
}
```

B. R code for the Pollard p-1 algorithm

```
library("FRACTION")
#install.packages("FRACTION")

pollardPminusOne <- function(n, B = 4){
  if(n < 3){
    return("Please enter a value that is larger than 2.")
    break
  }
  else{
    if(n %% 2 != 0){
```

```
    a = 2
  }
  else{
    a = floor(sqrt(n))+1
  }
  m = sieveOfEratosthenes(B)[length(sieveOfEratosthenes(B))]
  if( m == B){
    primes = sieveOfEratosthenes(B)[-length(sieveOfEratosthenes(B))]
  }else{
    primes = sieveOfEratosthenes(B)
  }
  for(i in 1:length(primes)){
    p = primes[i]
    e = 1
    while(p^e < B){
      primes[i] = p^e
      e = e+1
      if(((p^e) < B) | ((p^e) == B)){
        primes[i] = p^e
      }
    }
  }
  counterOne = 0
  counterN = 0
  for(i in 1:length(primes)){
    b = (a^primes[i]) %% n
    f = gcd((b-1),n)
    if((1 < f) & (f < n)){
      return(f)
      break
    }else if(f==1){
      counterOne = counterOne + 1
    }else if(f==n){
      counterN = counterN + 1
    }else{
      return("Failure")
      break
    }
    }
    if(counterOne == length(primes)){
      pollardPminusOne(n, B = B+1)
    }else{
      return(f)
    }

  }
}
```

C. R code for testing the Pollard p-1 function

```
pollardPminusOne(1)
pollardPminusOne(2)
pollardPminusOne(3)
pollardPminusOne(4)
pollardPminusOne(5)
pollardPminusOne(6)
pollardPminusOne(7)
pollardPminusOne(8)
pollardPminusOne(9)
pollardPminusOne(10)
pollardPminusOne(12)
pollardPminusOne(27)
pollardPminusOne(91)
pollardPminusOne(119)
pollardPminusOne(437)
pollardPminusOne(667)
pollardPminusOne(899)
```

D. Handout

Numerical Introductory Seminar - Prime Factorization

Anna Friesen

Motivation:	The cryptosystem RSA multiplies two large primes to create one part of the keys
Example:	Trial Division (inefficient for large prime factors)
Modification:	Test in Trial Division only primes
Sieve of Eratosthenes:	Creates list of primes (and zeros)

Code in Mathematica (Sieve of Eratosthenes)

```
SoE[n]:=Module[{v = Table[t,{t,1,n}],y = 2,k},
v = ReplacePart[v,0,1];
While[y * y ≤ n,k = y + y;
     While[k ≤ n,
          v = ReplacePart[v,0,k];
          k = k + y];
     y = y + 1];
     Return[v]]
```

Pollard (p-1) factorization method

Theoretical base:	Fermats little theorem
Previous algorithm:	Pollard rho method

Algorithm (Pollard (p-1) factorization method)

```
1. Input: $n \geq 2$
2. choose $a$ with $1 \leq a \leq n-1$ and arbitrary $B \in \mathbb{N}$
3. $\forall q \in \mathbb{P}, q \leq B$:
     • $a := a^q \pmod{n}$
     • $p := gcd(a-1, n)$
     • if $p \mid n$ break
     • else select new $a$ and go to step 3
4. return $p$
```

Ich erkläre hiermit, dass ich die Hausarbeit mit dem Titel *Prime Factorization* im Rahmen *des Kurses Numerical Introductory Seminar* im *Wintersemester 2018/2019* selbständig angefertigt, keine anderen Hilfsmittel als die im Literaturverzeichnis genannten benutzt und alle aus den Quellen und der Literatur wörtlich oder sinngemäß übernommenen Stellen als solche gekennzeichnet habe.

Berlin, den 12. Februar 2019
Anna Friesen