RSA

Arrow Algorithm and Modular Exponentiation

090832 Magnus Øverbø

Date:	October 4, 2015
Version:	1.0
Supervisor:	Slobodan Petrovic
Page count:	9
Word count:	
Appendices:	1
Keywords:	RSA, cryptology, public key cryptography, base conversion, arrow algorithm, modular exponentiation

IMT4532 — Cryptology 1

Department of Informatics and Media Technology
Gjøvik University College

1 Introduction

The arrow algorithm and modular exponentian provided in this report, can be used during the encryption / decryption of the RSA algorithm where one has the formula "$C = M^e \mod n$" for encryption and "$M = C^d \mod n$" for decryption.

The program was based on the slides provided by Slobodan Petrovic [1], the textbook proposed by Slobodan Petrovic for the course [2], Christof Paars textbook [3] and Williams Stallings textbook [4].

2 Program

The following sections explains the program in detail and the implementation of the arrow algorithm and the algorithm for calculating the modular exponentiation. To ease the process of programming the script, it has been divided into three separate parts.

1. Arrow algorithm for conversion to binary
2. Calculate base numbers
3. Calculate the modular exponentiation

2.1 Running the script

As shown in the code, the script is initiated by executing the command in one of the following ways "./MAIN.PY 2 1234 789" or "PYTHON ./MAIN.PY 2 1234 789". This will run the script using the three provided numbers as variables. The first argument passed is the base number of the expression to calculate, the second is the exponential value, and the third is the value to use for the modulus operation.

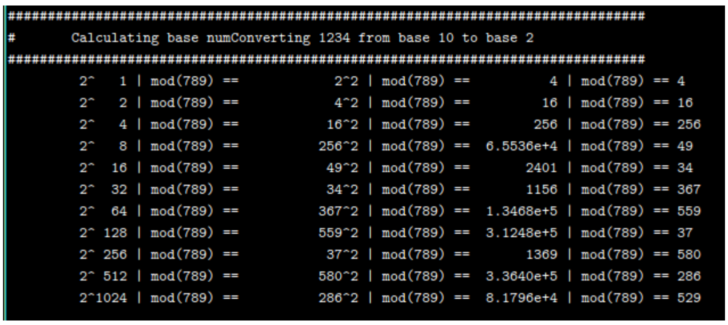
Eg. the mathematical expression, $x = 38^{75} \mid \mod (103)$ would for instance be executed by the script using the following command, "./MAIN.PY 2 75 103".

2.2 Software and algorithm explanations

The program is a basic script written in Python v2.7.6, but should work on Linux, OSX and Windows, and be independent of Python version v2.7 or v3.0.

pyCryptoAAME is written with the target of doing the calculation in two separate operations which then forms the basis for the third operation which calculates the final modular exponential value.

All calculations is performed in the script and visualised in the form of printing calculations to the standard output of the command line interface (CLI). See figure 1



```
################################################################################
#       Calculating base numConverting 1234 from base 10 to base 2
################################################################################
        2^   1 | mod(789) ==           2^2 | mod(789) ==         4 | mod(789) == 4
        2^   2 | mod(789) ==           4^2 | mod(789) ==        16 | mod(789) == 16
        2^   4 | mod(789) ==          16^2 | mod(789) ==       256 | mod(789) == 256
        2^   8 | mod(789) ==         256^2 | mod(789) ==  6.5536e+4 | mod(789) == 49
        2^  16 | mod(789) ==          49^2 | mod(789) ==      2401 | mod(789) == 34
        2^  32 | mod(789) ==          34^2 | mod(789) ==      1156 | mod(789) == 367
        2^  64 | mod(789) ==         367^2 | mod(789) ==  1.3468e+5 | mod(789) == 559
        2^ 128 | mod(789) ==         559^2 | mod(789) ==  3.1248e+5 | mod(789) == 37
        2^ 256 | mod(789) ==          37^2 | mod(789) ==      1369 | mod(789) == 580
        2^ 512 | mod(789) ==         580^2 | mod(789) ==  3.3640e+5 | mod(789) == 286
        2^1024 | mod(789) ==         286^2 | mod(789) ==  8.1796e+4 | mod(789) == 529
```

Figure 1: Output provided by the calculating the base numbers

2.2.1 Arrow Algorithm calculation

The arrow algorithm takes an integer and converts it to any base of choosing, this program converts it to base 2, which is the fixed value of "K" in the script. As shown in figure 2 the arrow algorithm is visualised to ease the users comprehension of the algorithm.

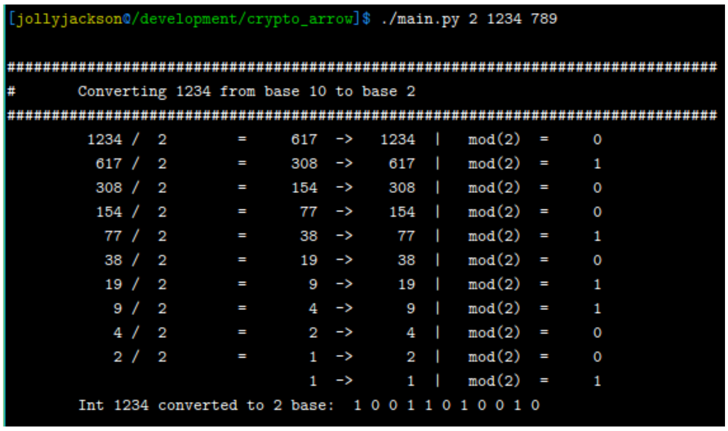


Figure 2: Output of arrow algorithm

The AA works by continuously dividing the number by "2" until it reaches "0". If there is a remainder in the calculated number, the calue is a binary "1", otherwise it is a "0". Anyways the number is rounded down to 0 decimals and repeated until it is finished at "0" or "1".

Code 2.1: Converion to binary

```
##########################################################
## Returns a list of "num"s bin val in rev order
##########################################################
def getBinary( num, bi ):
  if num == 0 or num == 1:          #If last run value is 0 / 1
    bi.append( num )                #Append the value to binary representation
    print "%35s  ->%7s  |   mod(%d)  = %5d" % ( str(num), str(num), K, num )
    return bi                       #Return the finished binary sequence

  x = num % K                       #Grab the remainder, just using modulo operand
  bi.append( x )                    #Append binary value to list
                                    #Print the calculation
  print "%13d / %2d %8s %7s  -> %6s  |   mod(%d)  = %5d" % ( num, K, "=",
      str(num/a), str(num), K, x )

  bi = getBinary( (num/K), bi )     #Recursive call to create the bin sequence
  return bi
```

2.2.2 Calculate Base numbers

This function produces a list containing all the base numbers to be used by the modular exponention function. Going through the binary representation of the exponent it calculates the next value based on the previous value or the initial value. For the first round the base number is set to "$B_1 = a^2 \mod n$", later iterations are calculated using the equation $B_i = {B_{i-1}}^K \mod n$. This is shown in figure 1.

These values are used for later processing when calculating the modular exponentiation.

Code 2.2: Calculate base numbers

```
########################################################
##  Calculates all base numbers for use in modExp
########################################################
def calcBase( I ):
  global base
  if I >= len(binary):                          #Cutoff function to finish calc
    return                                      #return to escape function
  elif I == 0:                                  #First calculation
    base.append( (a**K) % n)                    #append a^2 mod(n) as first val
    print "%10d^%4d | mod(%d) == %12d^%d | mod(%d) == %10s | mod(%d) == %d" % \
    (K, (K**I), n, a, K, n, sciNum(base[I]), n, base[I] )
  else:                                         #If not finished
    base.append( (base[I-1] ** K) % n )         #Add congruence value
    print "%10d^%4d | mod(%d) == %12d^%d | mod(%d) == %10s | mod(%d) == %d" % \
    (K, (K**I), n, base[I-1], K, n, sciNum(base[I-1]**K), n, base[I] )
  calcBase( I+1 )
```

2.2.3 Modular exponention

The basic gesture for modular exponentiation is to calculate the expontential value and performing the modulo operation on the product afterwards. This occurs however only when the binary representation is a "1".

The script has two starting functions, if the first bit of the representation is "0" the sum is set to "1" to avoid a "0*x" situation. Otherwise it is set to the initial value of $1^a \mod n$, where a is provided as an argument to the script.

For the remainding set where the binary is a "1", the final value is calculated as $d * e \mod n$ where "d" is the previous calculated mod-exp value and "e" is the previous base value.

When done the final value calculated is the modular exponentiation value.

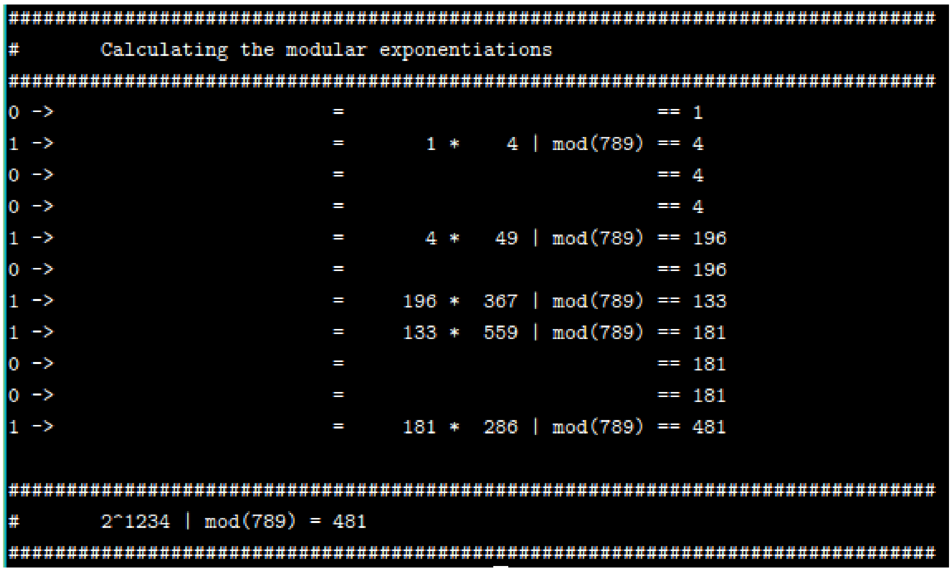


Figure 3: Output provided by the calculating the base numbers

Code 2.3: Calculate modular exponentiation

```
########################################################
##  Recursive function to calculate the mod exp
########################################################
def calcModExp( I ):
  global modExp                                 #Global values to use
  if I >= len(binary):                          #Cutoff function to finish calc
    return                                      #return to escape function
  elif I == 0:                                  #If first calculation
    if binary[I] == 1:                          #If calculation should be made
      modExp.append( (1 * a ) % n)              #Static calculation
      print "%d -> %6d^%4d | mod(%d) = %7d * " \
            "%4d | mod(%d) == %d" % (binary[I],
                a, b, n, 1, a, n, modExp[0])    #print first calc line
    if binary[I] == 0:                          #If calculation should be made
      modExp.append( 1 )              #Static calculation
      print "%d -> %22s = %25s == %d" % (binary[I], " ", " "  , modExp[I])

  else:                                         #If not finished
    d = modExp[I-1]                             #Prev modExp
    e = base[I-1]                               #Prev base value

    if binary[I] == 1:                          #If calculation should be made
      f = (d * e)                               #new modExp value
      g = f % n                                 #new modExp value
      modExp.append( g )                        #Add modExp value to list

      print "%d -> %22s = %7d * %4d | mod(%d) == %d" %(binary[I], " ", d,e,n,g)
    else:                                       #Print intermediate line
      print "%d -> %22s = %25s == %d" % (binary[I], " ", " "  , modExp[I-1])
      modExp.append( modExp[I-1] )              #Add previous modExp value

  calcModExp( I+1 )                             #Recursive call to nex calc
```

2.2.4 Helper functions

There are two functions created solely for the purpose of doing some work in order to print and manage the output of the script to the user.

First is the title(...) function which prints a frame around a short text, in order to use it as a title and divider between operations.

The second helper function was created to print the base value numbers x^2 during the base calculation. This prints numbers in a scientific manner as eg. "1,203 E+10".

Bibliography

[1] Slobodan Petrovic. Session 4 - asymmetric ciphers. Fronter.com, 2015. `https://fronter.com/hig/links/files.phtml/1928408341$798746276$/Lectures/Session_4_2015.pdf`, Verified: 06.09.2015.

[2] Lawrence Washington Wade Trappe. *Introduction to Cryptography - with Coding Theory*. Pearson International Edition. Pearson Education, 2nd edition, 2006. ISBN 0-13-198199-4.

[3] Jan Pelzl Christof Paar and Bart Preneel. *Understanding Cryptography: A Textbook for students and practitioners*. Springer Berlin Heidelberg, nov 2009. ASIN: B00HWUO98A.

[4] William Stalling. *Cryptography and Network Security: Principles and Practices*. International Edition. Pearson, 6 edition, 2014. ISBN-13: 978-0-273-79335-9.

A Scripts and Source codes

A.1 Complete Python Script

Code A.1: Complete Python Script

```
#!/usr/bin/env python
#######################################################
# Stud nr:  090832
# Date:     05.09.2015
#
# Project:  RSA
# Program:  Arrow Algorithm and Modular exponentiation
# Descr:    Implementation of the arrow algorithm
#           with the use of modular exponentiation
#######################################################

######################################################
##  Imports
######################################################
import sys

######################################################
##  Variables
######################################################
binary  = []    #Binary holder for b, the reverse binary sequence of b
modExp  = []    #Modulated calculations, holds the output val of mod exp calcs
base    = []    #Base value holder, n-1 length array

a       = 0     #Integer base value
b       = 0     #Integer exponential value
n       = 0     #Integer modulus value
ML      = 5     #Max integer length for printing scientific numbers
K       = 2     #Fixed value to use as exponential value and divisor

######################################################
##  Input parameter check
######################################################
if len( sys.argv ) != 4:
  print "Execute program with the following command:\n\t"      \
        "./main.py [base (a)] [exponent (b)] [mod (n)]\n"       \
        "Eg. './main 2 1234 789', should output 481 as "        \
        "shown on p78-79 in Trappe and Washinton 2nd Ed\n"
  exit()

try:                          #Try to convert input arg to integer values
  a = int( sys.argv[1] )      #Grab base number of expresison from argument list
  b = int( sys.argv[2] )      #Grab the exponential value from argument list
  n = int( sys.argv[3] )      #Grab the modulus value from argument list
except:                       #If conversion to integer failse print error
  print "Incorrect input, try again ['%s', '%s', '%s']" % (sys.argv[1],
      sys.argv[2], sys.argv[3])
  exit()



```

```
#######################################################
##  Helper function to print headings
#######################################################
def title( d, n, out ):
  print "\n%s\n%s\t%s\n%s" % (d*n, d, out, d*n) #Print divisor, string, divisor

#######################################################
##  Helper function to print large numbers
#######################################################
def sciNum( c ):
  if c < 10000:
    return str( c )

  s = str( c )                    #Convert number to string
  t = len(s)-1                    #number of e
  l = list(s[:ML])                #Convert 5 first numbers to list
  l.insert(1, ".")                #Add a comma after first number

  o = ""
  if t > 0:                       #If org number is more than 10
    o = "".join(l) + "e+%d"  % t  #Create scientific number
  else:
    o = "".join(l) + "0e+%d" % t  #Add after comma when appending

  return o

#######################################################
##  Returns a list of "num"s bin val in rev order
#######################################################
def getBinary( num, bi ):
  if num == 0 or num == 1:        #If last run value is 0 / 1
    bi.append( num )              #Append the value to binary representation
    print "%35s  ->%7s  |   mod(%d)  = %5d" % ( str(num), str(num), K, num )
    return bi                     #Return the finished binary sequence

  x = num % K                     #Grab the remainder, just using modulo operand
  bi.append( x )                  #Append binary value to list
                                  #Print the calculation
  print "%13d / %2d %8s %7s  -> %6s  |   mod(%d)  = %5d" % ( num, K, "=",
      str(num/a), str(num), K, x )

  bi = getBinary( (num/K), bi )   #Recursive call to create the bin sequence
  return bi


#######################################################
##  Calculates all base numbers for use in modExp
#######################################################
def calcBase( I ):
  global base
  if I >= len(binary):                          #Cutoff function to finish calc
    return                                      #return to escape function
  elif I == 0:                                  #First calculation
    base.append( (a**K) % n)                    #append a^2 mod(n) as first val
    print "%10d^%4d | mod(%d) == %12d^%d | mod(%d) == %10s | mod(%d) == %d" % \
    (K, (K**I), n, a, K, n, sciNum(base[I]), n, base[I] )
  else:                                         #If not finished
    base.append( (base[I-1] ** K) % n )         #Add congruence value
    print "%10d^%4d | mod(%d) == %12d^%d | mod(%d) == %10s | mod(%d) == %d" % \
    (K, (K**I), n, base[I-1], K, n, sciNum(base[I-1]**K), n, base[I] )
  calcBase( I+1 )

```

```python

##########################################################
##  Recursive function to calculate the mod exp
##########################################################
def calcModExp( I ):
  global modExp                                 #Global values to use
  if I >= len(binary):                          #Cutoff function to finish calc
    return                                      #return to escape function
  elif I == 0:                                  #If first calculation
    if binary[I] == 1:                          #If calculation should be made
      modExp.append( (1 * a ) % n)              #Static calculation
      print "%d -> %6d^%4d | mod(%d) = %7d * " \
            "%4d | mod(%d) == %d" % (binary[I],
                a, b, n, 1, a, n, modExp[0])    #print first calc line
    if binary[I] == 0:                          #If calculation should be made
      modExp.append( 1 )              #Static calculation
      print "%d -> %22s = %25s == %d" % (binary[I], " ", " "  , modExp[I])

  else:                                         #If not finished
    d = modExp[I-1]                             #Prev modExp
    e = base[I-1]                               #Prev base value

    if binary[I] == 1:                          #If calculation should be made
      f = (d * e)                               #new modExp value
      g = f % n                                 #new modExp value
      modExp.append( g )                        #Add modExp value to list

      print "%d -> %22s = %7d * %4d | mod(%d) == %d" %(binary[I], " ", d,e,n,g)
    else:                                       #Print intermediate line
      print "%d -> %22s = %25s == %d" % (binary[I], " ", " "  , modExp[I-1])
      modExp.append( modExp[I-1] )              #Add previous modExp value

  calcModExp( I+1 )                             #Recursive call to nex calc



##########################################################
##  Main running function
##########################################################
title( "#", 80, "Converting %d from base 10 to base %d" % (b,K))
binary = getBinary( b, [] )           #calculate the reverse binary sequence
                                      #Print bin seq in correct order high->low
print "\tInt %d converted to %d base: " % ( b, K),
for x in binary[::-1]:
  print x,
print

title("#",80, "Calculating base numConverting %d from base 10 to base %d"%(b,K))
calcBase( 0 )

title( "#", 80, "Calculating the modular exponentiations")
calcModExp( 0 )                 #Start to calculate the mod exponentials


title( "#",80, "%d^%d | mod(%d) = %d" % (a, b, n, modExp[len(modExp)-1] ) )
```