

*Library copy*

*3/3/98*

# SOFTWARE IMPLEMENTATION OF RSA ALGORITHM

Dertation submitted in partial fulfilment of the requirements for the  
award of the degree of

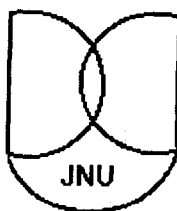
**MASTER OF TECHNOLOGY**

in

**COMPUTER SCIENCE**

by

**BRAHMA PRAKASH**



**SCHOOL OF COMPUTER & SYSTEM SCIENCES  
JAWAHAR LAL NEHRU UNIVERSITY  
NEW DELHI - 110067  
January 1998**

*TH*

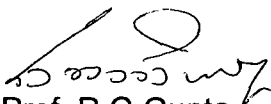
005.1 TH  
'8848 So

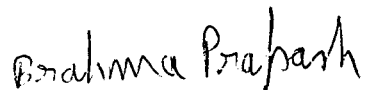


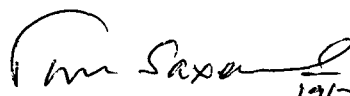
TH6846

## CERTIFICATE

This is to certify that dissertation entitled SOFTWARE IMPLEMENTATION OF RSA ALGORITHM being submitted by BRAHMA PRAKASH to the School of Computer & System Sciences of the JAWAHAR LAL NEHRU UNIVERSITY, NEW DELHI in the partial fulfilment of the requirements for the award of the degree of Master of Technology in Computer Science is a bonafide work carried out by him under the guidance and supervision of Prof. R.G.Gupta. The matter embodied in the dissertation has not been submitted for the award of any other degree or diploma.

  
Prof. R.G.Gupta  
Professor, SC&SS  
Jawaharlal Nehru University  
New Delhi 110067

  
Brahma Prakash

  
Prof. P.C.Saxena 19/12  
Dean, SC&SS  
Jawaharlal Nehru University  
New Delhi 110067



## **ACKNOWLEDGEMENT**

I am very glad to express my sincere gratitude to my guide, Prof. R.G.Gupta for his valuable guidance in completing this project successfully. He provided me important literature and books related to this project. He always inspired me. I feel it is a great privilege to have had the opportunity to work under his prestigious supervision. His constant encouragement and support helped me immensely. He also rectified my personality.

Prof. Gupta arranged to attend me a course no. MA531N "Cryptology" co-ordinated by Dr. V.Ch.Venkaiah, in IIT Delhi from July to November 1997. I am also thankful to Dr. V.Ch.Venkaiah, who gave me opportunity to attend his lecture classes and gave beneficial suggestions and advice to improve this project.

I am also thankful to my friends who helped me in carrying out this project.

Date:

**BRAHMA PRAKASH**

# CONTENTS

CERTIFICATE	i
ACKNOWLEDGEMENT	ii
ABSTRACT	iv
1- INTRODUCTION	1
1.1 Secure communication	2
1.2 Cryptography	3
1.3 Public key cryptography	5
2- RSA ALGORITHM	8
2.1 Key generation	9
2.2 Encryption and Decryption	13
3- MULTI-PRECISION ARITHMETIC	17
3.1 Addition	18
3.2 Subtraction	19
3.3 Multiplication	20
3.4 Division	21
3.5 Modular reduction	23
3.6 Fast exponentiation	27
4- DESIGN & IMPLEMENTATION	33
5- CODING & TESTING	36
6- COMPARISON	38
APPENDIX - A: Source code	39
APPENDIX - B: Test data & its output	50
APPENDIX - C: Bibliography	52

## **ABSTRACT**

In development of this software the classic life cycle paradigm is used. Sometimes it is called the "waterfall model". The life cycle paradigm demands a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing and maintenance. Following are the description of the steps.

**System engineering and analysis:** For implementing RSA algorithm, work begins by establishing requirements for all basic issues and elements and then allocating these requirements to develop a software. This view is essential when one must interface with other elements of software. System engineering and analysis encompasses requirements gathering at the system level with a small amount of top-level design and analysis.

**Software requirement analysis:** The requirements gathering process is intensified and focused specifically on software. To understand the nature of the program to be built, information domain must be understood for the software as well as the required function and performance. Requirements are documented and reviewed through the problem.

**Design:** Software design is actually a multistep process that focuses on four distinct attributes of the program: data structure, software architecture, procedural detail, and interface characterization. The design

process translates requirements into a representation of the software that can be assessed for quality before coding begins.

**Coding:** The design must be translated into a machine readable form. The coding step performs this task. If design is performed in a detailed manner, coding can be accomplished mechanically.

**Testing:** Once code has been generated, program testing begins. The testing process focusses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals, that is, conducting tests to uncover errors and ensure that defined input will produce actual results. In this step sometimes it may happen that, we have to correct a function which is tested before and found correct, but when it is in use in another function and this function gives some errors due to error of function called.

**Maintenance:** Software will undoubtedly undergo change after its development because of the functional or performance enhancements. Software maintenance reapplies each of the preceding life-cycle steps to an existing program rather than a new one.

In chapter 1, Need of secure communication, necessity, benefits etc. are explained. Theme of cryptography (private & public) is explained and some well known public key cryptosystem is given.

In chapter 2, RSA algorithm is explained. Key generation, properties & generation of prime numbers, computing gcd & modulo inverse are explained. Encryption and Decryption have been explained.

In chapter 3, Multi-precision arithmetic is taken as a primary issue which is the basic building block of computations in RSA algorithm. In this chapter we have considered addition, subtraction, multiplication and division. We also explained modular reduction and fast modular exponentiation algorithms.

In chapter 4, we have explained design and implementation of some ideas which are helping to implement some different algorithms successfully.

In chapter 5, coding and testing are explained. Program metric and software metric, execution times etc. are given in this chapter. Source code and test datas are given in appendices A and B respectively.

In chapter 6, there is comparison between RSA cryptosystem and RSA based cryptosystem. We have compared it on the basis of speed, complexity and security measure.

## CHAPTER 1

### INTRODUCTION

Communication is taking place in day-to-day life of every one. In defence and political sphere, it is very important to maintain confidentiality of information. RSA algorithm may perform this role. Popularity and efficiency of RSA algorithm makes it in use in electronic commerce, electronic banking, electronic fund transfer etc. Computer networking is spreading very fast around the globe. RSA algorithm may be used in these computer networks. It serves the purpose of authentication, digital signatures etc. Processing speed of RSA algorithm is slow. It is to be equalize with high data bit rate communication channels. It may perform wonder in future.

In 1977 Ronald Rivest, Adi Shamir and Leonard Adleman took up the challenge of producing a full-fledged public-key cryptosystem. The process lasted several months during which Rivest proposed approaches, Adleman attacked them, and Shamir recalls doing some of each. In May 1977 they were rewarded with success. They had discovered how a simple piece of classical number theory could be made to solve the problem. Implementation of RSA algorithm was a challenging job. H/W & S/W implementation of RSA had achieved by some people. VICTOR was an efficient RSA H/W implementation. But S/W implementation of RSA was not efficient. It took long processing time while communication channels can carry information very fast.



Two large prime integers are chosen and multiplication of these is published in public directory as a public number. Euler's totient function is computed. Secret key is chosen and tested for coprime of totient function. Public keys are derived from the secret keys and totient function computing multiplicative inverse of totient function and published in a public directory. Ciphertext is computed as plaintext to the power public key modulo public number and plaintext is recovered from ciphertext as ciphertext to the power secret key modulo public number.

Number theory is the root of the RSA algorithm. Factorization of very large number is very much time consuming, it is the strength of RSA algorithm. We need Multi-precision arithmetic. Modulus is being computed using modular reduction algorithm which is fast. Exponentiation uses square-and-multiply technique, right to left binary technique, k-SR, k-ary, SS(l) etc. which make very fast computation of modular exponentiation. Some method is used at implementation level to make this software fast.

### **1.1 Secure Communication**

Secure communication activities involve exchange of information from a person to another person maintaining confidentiality. Communication activities may be performed through satellites, microwaves, radiowaves, digital communication systems, electronic communication systems etc. Secure communication activities are needed in the field of defence, diplomatic activities, business, share market,

political sphere. Secure communication can be achieved through public key cryptosystem which also serves the purpose of authentication, authorization, digital signatures etc. Therefore we can say that nowadays one can communicate with another securely.

## **1.2 Cryptography**

Cryptography is the science of secret communications which involves controlled corruption of messages or signals so that only authorized person may be able to make meaning out of them. Historically people belonging to four different part of life have used and contributed to the art of cryptography: the military, the diplomatic corps, the diarists and lovers. Of these, military has had the most important roles. For exchange of secret information in the defence, diplomatic corps, business, scientific research and other secret informations of the interest of national security we need some safe method of communication. A large volume of data is exchanged and communicated through internet. e-mails also need secure propagation. Home pages and Websites need authorization. Some may trap these datas and may perform illegal access if there is no prevention. For preventing illegitimate access cryptography can be used.

Cryptography can be classified into two streams. One is private key cryptography and another is public key cryptography. In private key cryptography, key is secret and same for both ciphering and deciphering i.e. keys are symmetric. In public key cryptography there is a public key for

ciphering which is known to all because it is published in a public directory and there is a secret key for deciphering which can not be obtained knowing public key & method used i.e. here keys are asymmetric.

Private key cryptography: Private key cryptography was used earlier when computers were not commonly available. There are various private key cryptosystem e.g. substitution cipher, transposition cipher etc. Since computers came in existence, it is very easy to break the long messages using statistical methods. Using statistical distribution of English alphabet one can break the substitution cipher for long messages. There are also problems of key distribution and maintenance. As number of users increases number of keys increases quadratically. It does not serve the purpose of authentication, authorization, digital signatures. It may not be effective for secure communication. The only advantage of this is that it can cipher and decipher very fast, at the rate of the communication channels. Public key cryptography overcome all these problems but it is a slow process.

Public key cryptography: Since computer systems came in existence, public key cryptography became more popular. In this method there are public and private keys. Public keys are known to all but private keys are known to only receivers. Sender and receiver have different private keys. They don't know private key of each other. It also serves law enforcement. One can claim in the court on the basis of it. That is why it is applied in electronic commerce, electronic banking, electronic fund transfer and

various business applications. It may not be broken easily that is why it is applied in defence, diplomatic corps, political sphere, scientific research etc. Different public key algorithms are there e.g. Elgamal's, Deffi-Hellman's key exchange protocol, Merkle-Hellman's knapsack trapdoor algorithm, RSA algorithm etc. The only problem with public key cryptography is slow processing. To overcome this problem various techniques are used but it is not serving our purpose completely. Various techniques are under development process that will enable it to face high data bit rate communications and there will be no delay in processing, that is, flow will be maintained. It needs fast multi-precision arithmetic operations.

### 1.3 Public Key Cryptography

It is very hard to compute secret keys, knowing public key, is the theme around the public key cryptography. Some public key algorithms are given below.

(a) Diffie-Hellman's key exchange protocol

Secret key:  $x_A$   $x_B$   
 Public Key:  $p$   
 $g$  is chosen by both users.

A	B
$g^{x_A} \pmod{p}$	$g^{x_B} \pmod{p}$
$g^{x_A x_B} \pmod{p}$	$g^{x_A x_B} \pmod{p}$

(b) Merkle-Hellman's knapsack trapdoor algorithm

$A=(a_1 a_2 a_3 \dots a_n)$  where  $a_i > 0$

It is a super increasing sequence i.e.  $a_i > a_j$  ( $j=1..i-1$ )  
 $p$  is chosen such that  $p > \sum a_i$  and  $w$  is chosen such that  $\gcd(w,p)=1$ .

secret key:  $A, p, w$

Public key:  $A'=(a_1*w(\text{mod } p), a_2*w(\text{mod } p), \dots a_n*w(\text{mod } p))$ .

$M=(m_1, m_2, \dots, m_n)$  where  $m_i \in \{0, 1\}$ .

$A'M=C'$  where  $C'$  is a positive integer.

$w'C'(\text{mod } p)=C$

$AM=C$

Multiple Iteration:  $A=(a_1, a_2, \dots, a_n)$  where  $a_i > 0$

$$\begin{aligned} &w_1 p_1 \\ &A'=A*w_1(\text{mod } p_1) \\ &w_2 p_2 \\ &A''=A'*w_2(\text{mod } p_2) \\ &w_2'C''(\text{mod } p_2)=C' \\ &w_1'C'(\text{mod } p_1)=C \end{aligned}$$

(c) Elgamal's cryptosystem: It is based on discrete logarithm problem.

$(\alpha^k, m\alpha^{xak})$  : encrypted

$$\begin{aligned} &A \\ &\alpha^{xA}, p, \alpha \end{aligned}$$

$m$  (message)

$B \rightarrow A$

$k$

$$(\alpha^k, m_1 \alpha^{xak})$$

$$(\alpha^k)^{xA} \rightarrow \alpha^{kxA}$$

$$(\alpha^k, m_2 \alpha^{xak})$$

$$m_1 \alpha^{xak} = C_1$$

$$m_2 \alpha^{xak} = C_2$$

$$\begin{aligned} \frac{C_1}{C_2} &= \frac{m_1}{m_2} \end{aligned}$$

(d) RSA algorithm:

$p, q$ : large prime integer of the order of 100 digit.

$$n = p \cdot q$$

$$\phi(n) = (p-1)(q-1)$$

$$\gcd(d, \phi(n)) = 1$$

$$e \cdot d = 1 \pmod{\phi(n)}$$

$$a^e \pmod{n} = c$$

$$c^d \pmod{n} = a$$

(e) RSA based public key cryptosystem:

choose  $p_i$  &  $g_j$  where  $1 \leq i \leq 3$ ,  $1 \leq j \leq 6$

$p_i < p_j$  whenever  $i < j$

$$\gcd(p_i, g_j) = 1 \quad \forall i$$

Encryption:

$$c_i = p_i^e \pmod{N} \quad \forall i$$

$$c_{3+j} = g_j^e \pmod{N} \quad \forall j$$

$$c_{9+i} = (((m_i + g_4)g_1 \pmod{p_1} + g_5)g_2 \pmod{p_2} + g_6)g_3 \pmod{p_3}$$

Decryption:

$$c_i^d \pmod{N} = p_i^{ed} \pmod{N} = p_i \quad i=1,2,3$$

$$c_{3+j}^d \pmod{N} = g_j^{ed} \pmod{N} = g_j \quad j=1, \dots, 6$$

$$m_i = ((((((c_{9+i} g_3^{-1} \pmod{p_3} - g_6) \pmod{p_3}) g_2^{-1} \pmod{p_2} - g_5) \pmod{p_2}) g_1^{-1} \pmod{p_1} - g_4) \pmod{p_1})$$

## CHAPTER 2

### RSA ALGORITHM

RSA is most popular public key algorithm. It is named for its inventors Ronald Rivest, Adi Shamir and Leonard Adleman. It is based on the fact that current computing art of factorization of composite numbers with large prime factors involve overwhelming computations. Multi-precision integer arithmetic is used in RSA algorithm. It is secure system because breaking of RSA is easy by method but very much time consuming (of the order of millions of years). RSA algorithm involves following assumptions and computations.

Two very large prime integers are chosen as private keys which are nearly equal. Multiplication of these two prime numbers is public. Euler's totient (also called indicator,  $\phi$ ) function of it is computed and is private. A number which is coprime of totient function is chosen and it is private. Multiplicative inverse of it modulo totient function is computed and it is public. Ciphertext is computed as plaintext to the power public key modulo multiple of primes.

Deciphering is carried out by computing ciphertext to the power private key modulo multiple of primes. The fact that how plaintext will be recovered from ciphertext is based on that multiplication of both private & public key modulo totient function is equivalent to integer one.

Multiplicative inverse is computed by using Euclid's algorithm. Euclid algorithm computes greatest common divisor of two integers. We choose

secret key and if gcd of it w.r.t. totient function is not one, then we divide chosen number by gcd, then gcd of this number and totient function will be one. Then we compute inverse. For computing exponent we can use repeated square and multiply technique. We are describing methods for key generation and Encryption & Decryption.

## 2.1 Key generation

We are explaining prime numbers and Euclid's algorithm which are used to generate keys for our cryptosystem.

a) Prime number generation: We used given theorem to generate required prime number. It may happen that generated number is psuedo prime yet it will fulfil our requirement.

Theorem: If  $\gcd(a,p)=1$  and  $a^{p-1} \pmod p=1$  then p is a prime number.

Using this theorem it is very easy to check that a given number is prime or not. We can choose an odd number and then we test this property for that number if it is satisfied then the number is prime otherwise we subtract or add two in that number and test this property again untill the number is not prime. Some properties and other information about prime numbers are given.

Distribution of primes: A computation by M. Kraitchik shows that for each group of 100 numbers in the interval from  $10^{12}+1$  to  $10^{12}+1000$  the corresponding frequency of primes is

4,6,2,4,2,4,3,5,1,6



Even though the prime numbers gradually become more scarce as the numbers within the groups become larger, there are still infinitely many primes.

According to the prime number theorem, the ratio of  $\pi(x)$ , the number of primes in the interval from 2 to  $x$ , and  $x/\ln(x)$  approaches 1 as  $x$  becomes very large, that is.

$$\lim_{x \rightarrow \infty} \frac{\pi(x)}{x/\ln(x)} = 1$$

where  $\ln(x)$  is the (natural) logarithm of  $x$  to the base  $e=2.71828\dots$ . On the average about  $(x-1)/\pi(x)=\ln(x)$  values must be tested before a prime is found.

Testing for primality: Several methods can be used to test a randomly selected number for primality. However, the most straightforward approaches are not computationally feasible. For example, a test could be based on Wilson's theorem, which states that

$$(p-1)! \equiv -1 \pmod{p} \quad \text{if } p \text{ is prime}$$

where  $(p-1)! = 2 \cdot 3 \cdot \dots \cdot (p-1)$

In all other cases (except  $n=4$ ), it can be shown that

$$(n-1)! \equiv 0 \pmod{n} \quad \text{if } n \text{ is not prime}$$

It should be obvious, however, that a test based on Wilson's theorem is useless for large values of  $p$ , since too many multiplications would be required to compute  $(p-1)!$ .

A different test could be based on the simple fact that if a number  $n$  is not prime, then  $n$  must contain a factor less or equal to the square root of  $n$ . But even here the test is useless for large primes  $p$ , since to show that  $p$  is not divisible by any number between 2 and  $p$ , and thus prove that  $p$  is prime, would still require too many computations.

To test a large number  $n$  for primality, one could use the elegant "probabilistic" algorithm of Solovay and Strassen. It picks a random number  $a$  from a uniform distribution  $(1, 2, \dots, n-1)$  and tests whether

$$\gcd(a, n) = 1 \quad \text{and} \quad J(a, n) a^{(n-1)/2} \pmod{n} \quad (\text{eq 10})$$

where  $J(a, n)$  is the Jacobi symbol. if  $n$  is prime, then equation 10 always holds. If  $n$  is composite, the equation 10 will be false with probability of at least  $1/2$ .

The number  $n$  can now be tested for primality by using a set of integers,  $A = \{a_1, a_2, \dots, a_m\}$ , where each  $A$  is less than  $n$ . The test requires that, for each value of  $a$  in  $A$ , equation 10 holds. Thus  $n$  is found to be composite if there is an  $a$  in  $A$  for which equation 10 does not hold; otherwise  $n$  is accepted as prime.

The procedure does not guarantee that a selected number is prime, but only that it has not failed the test of primality. The greater the number of integers in  $A$ , the greater the probability that a selected number is prime.

When  $n$  is odd,  $a$  and  $n$  are coprime, and  $\gcd(a,n)=1$ , the Jacobi symbol,  $J(a,n)$ , has a value in  $\{-1,1\}$  and can be efficiently computed by the following recursive procedure:

$$J(a,n) = \begin{cases} 1 & \text{if } a=1 \\ J(a/2,n)(-1)^{(n^2-1)/8} & \text{if } a \text{ is even} \\ J(n \pmod a, a)(-1)^{(a-1)(n-1)/4} & \text{else} \end{cases}$$

Prime numbers are generated, now we need to compute Public keys and Secret keys. Modulo public number is computed by multiplying two prime numbers. Secret keys and other public keys can be computed using Euclid algorithm given below.

b) Euclid Algorithm: Let  $a > b > 0$  be two integers. By division property

$$\begin{aligned} a &= b \cdot q_1 + r_1 & 0 < r_1 < b \\ b &= r_1 \cdot q_2 + r_2 & 0 < r_2 < r_1 \\ r_1 &= r_2 \cdot q_3 + r_3 & 0 < r_3 < r_2 \end{aligned}$$

$$\begin{aligned} r_{n-2} &= r_{n-1} \cdot q_n + r_n & 0 < r_n < r_{n-1} \\ r_{n-1} &= r_n \cdot q_{n+1} \end{aligned}$$

where  $r_n \neq 0$  but  $r_{n+1} = 0$

The gcd of  $a$  and  $b$  is then  $r_n$

If  $\gcd(a,b)=1$  then multiplicative inverse of  $b$  modulo  $a$  exist and computing it we need to define

$$\begin{aligned} y_1 &= 0 - 1 \cdot q_1, \\ y_2 &= 1 - y_1 \cdot q_2, \\ y_3 &= y_1 - y_2 \cdot q_3, \\ y_4 &= y_2 - y_3 \cdot q_4, \end{aligned}$$

$$y_n = y_{n-2} - y_{n-1} * q_n,$$

$$y_{n+1} = y_{n-1} - y_n * q_{n+1},$$

If  $rn=1$  then multiplicative inverse of  $b$  modulo  $a$  is the positive residue of  $y_n$  modulo  $a$ .

we can compute the multiplicative inverse as in the following table

a	0	
b	1	
$q_1$	$r_1$	$y_1$
$q_2$	$r_2$	$y_2$
$q_n$	$r_n$	$y_n$
$q_{n+1}$	$r_{n+1}$	$y_{n+1}$

## 2.2 Encryption & Decryption

To explain the Encryption and Decryption process we are explaining RSA algorithm.

### RSA algorithm:

- |                                 |             |
|---------------------------------|-------------|
| 1. $p$ and $q$ are primes       | (secret)    |
| 2. $r=p.q$                      | (nonsecret) |
| 3. $\phi(r)=(p-1)(q-1)$         | (secret)    |
| 4. SK is the private key        | (secret)    |
| 5. PK is the public key         | (nonsecret) |
| 6. X is the message (plaintext) | (secret)    |
| 7. Y is the ciphertext          | (nonsecret) |

The RSA algorithm is based on an extension of Euler's theorem, which states that

$$a^{\phi(r)} \equiv 1 \pmod{r}$$

where

1.  $a$  must be relatively prime to  $r$ .
2.  $\phi(r) = r(1 - 1/p_1)(1 - 1/p_2) \dots (1 - 1/p_n)$ , where  $p_1, p_2, \dots, p_n$  are the prime factors of  $r$ .

$\phi(r)$  is Euler's  $\phi$ -function of  $r$  ( also called indicator or totient ) which determines how many of the numbers  $1, 2, \dots, r$  are relatively prime to  $r$ .

To obtain the mathematical relationship between the public and private keys, PK and SK, Euler's result is extended as follows. First it is shown that  $a^b \pmod{r}$  implies that  $a^m \equiv b^m \pmod{r}$  for any exponent  $m$ . Thus Euler's formula  $a^{\phi(r)} \equiv 1 \pmod{r}$  can be rewritten as

$$a^{m\phi(r)} \equiv 1 \pmod{r} \quad (\text{eq 1})$$

where, as before,  $a$  is relatively prime to  $r$ . From the fact that  $a \equiv b \pmod{r}$  implies that  $ac \equiv bc \pmod{r}$  for any integer  $c$ , and from above equation it follows that

$$X^{m\phi(r)+1} \equiv X \pmod{r} \quad (\text{eq 2})$$

where plaintext  $X$  is relatively prime to  $r$  ( a restriction which will be removed later on ). Let the public key PK and secret key SK is chosen so that

$$\text{SK} \cdot \text{PK} = m\phi(r) + 1 \quad (\text{eq 3})$$

or, equivalently,

$$\text{SK} \cdot \text{PK} \equiv 1 \pmod{\phi(r)} \quad (\text{eq 4})$$

Equation 2 can be therefore rewritten as

$$X^{SK.PK} \equiv X \pmod{r} \quad (\text{eq 5})$$

which holds true for any plaintext ( $X$ ) that is relatively prime to the modulus  $\phi(r)$ .

Encipherment and decipherment can now be interpreted as follows:

$$E_{PK}(X) = Y \equiv X^{SK} \pmod{r} \quad (\text{eq 6})$$

$$D_{SK}(Y) \equiv Y^{PK} \pmod{r} \equiv X^{SK.PK} \pmod{r} \equiv X \pmod{r} \quad (\text{eq 7})$$

Moreover, because multiplication is a commutative operation (i.e.  $SK.PK = PK.SK$ ), it follows that encipherment followed by decipherment is equivalent to decipherment followed by encipherment:

$$D_{SK}(E_{PK}(X)) = E_{PK}(D_{SK}(X)) \equiv X \pmod{r} \quad (\text{eq 8})$$

This property is useful for generating digital signatures.

Because  $X^{PK} \pmod{r} \equiv (X + mr)^{PK} \pmod{r}$  for any integer  $m$ , each plaintext  $X, X+r, X+2r, \dots$ , results in the same ciphertext. Thus the transformation from plaintext to ciphertext is many-to-one. But restricting  $X$  to the set  $\{0, 1, \dots, r-1\}$  makes the transformation one-to-one, and thus encipherment and decipherment can be achieved as described in equation 6 and 7.

Theorem: Prove that  $X^{m\phi(r)+1} \equiv X \pmod{r}$  holds for any plaintext,  $X$ , where  $r=pq$  is the product of two prime factors and  $X$  is restricted to the set  $\{0, 1, \dots, r-1\}$  - a condition which is necessary for encipherment and decipherment.

proof: The theorem holds trivially for  $X=0$ , and so only the case  $X>0$  must be considered. If  $X$  is not relatively prime to  $r=pq$ , then  $X$  must contain either  $p$  or  $q$  as a factor. Suppose  $p$  is a factor of  $X$ , so that the relation  $X=cp$  holds for some +ve integer  $c$ . Since  $X$  is restricted to the set  $\{0,1,\dots,r-1\}$ , and  $r$  equals  $pq$ , it follows that  $X$  must be relatively prime to  $q$ . Otherwise,  $X$  would also contain  $q$  as a factor, in which case it would exceed  $r-1$ . Using Euler's theorem, we have

$$X^{\phi(q)} \equiv 1 \pmod{q}$$

where  $\phi(q)=q-1$ . But

$$X^{m(p-1)\phi(q)} \equiv 1^{m(p-1)} \equiv 1 \pmod{q}$$

for any integer  $m$ , and  $(p-1)\phi(q)=(p-1)(q-1)=\phi(r)$ , so that

$$X^{m\phi(r)} \equiv 1 \pmod{q}$$

or, for some integer  $n$

$$1 = X^{m\phi(r)} + nq$$

Multiplying each side by  $X=cp$  results in

$$X = X^{m\phi(r)+1} + (nq)(cp)$$

$$= X^{m\phi(r)+1} + ncr$$

or,  $X^{m\phi(r)+1} \equiv X \pmod{r}$

The case in which  $q$  is a factor of  $X$  can be handled in the same manner, thus completing the proof.

Encryption process:  $X^{PK} \pmod{n} = Y$

Decryption process:  $Y^{SK} \pmod{n} = X$

## CHAPTER 3

# MULTI-PRECISION ARITHMETIC

In this chapter we are describing basic operations (addition, subtraction, multiplications, division) and some more operations which are used in RSA algorithm.

Let us now consider operations on numbers which have arbitrarily high precision. For the sake of simplicity, let us assume that we are working with integers instead of numbers with an embedded radix point. We will also be working only with positive integers. Now we will discuss algorithm for:

- a) addition and subtraction of  $n$ -place integers, giving an  $n$ -place answer and a carry;
- b) multiplication of an  $n$ -place integer by an  $m$ -place integer, giving an  $(m+n)$ -place answer;
- c) division of an  $(m+n)$ -place integer by an  $n$ -place integer, giving an  $(m+1)$ -place quotient and an  $n$ -place remainder.

These algorithms for basic operations may be called "the classical algorithms", since the word "algorithm" was used only in connection with these processes for several centuries. By the term " $n$ -place integer", we mean any integer less than  $b^n$ , where  $b$  is the radix of the conventional positional notation in which the numbers are expressed; such numbers can be expressed using at most  $n$  "places" in this notation. The most



important fact to understand about multiple precision integers is that they may be regarded as numbers written in radix  $b$  where  $b$  could be the computers word size. For example if we have a computer with word size  $b=10^{10}$ , then an integer that fills up 10 words has actually 100 decimal digits; but we will consider it to be a 10 place integer to the base  $10^{10}$ . It is clear that the larger the base, the lesser will be the number of computer words required to store the number. Obviously the base cannot be larger than the size of the computer word.

Let us first state the primitive operation into which we will decompose the above problem:

a) addition and subtraction of one place integers, giving a one place answer and a carry:

b) multiplication of one place integer by another one place integer, giving a two place integer:

c) division of a two place integer by a one place integer, provided that the quotient is a one place integer, and yielding also a one place remainder.

By adjusting the word size, if necessary, nearly all computers will have these three operations available, and so we can construct our algorithms mentioned above in terms of these primitive operations. Let us now consider each of these algorithms in more detail.

### **3.1 Addition**

Addition is the simplest among all the operations described above, but some of the ideas involved occur in other algorithms also. The idea

involved in the algorithm is the usual pencil and paper method adopted from Knuth(volume II, semi-numerical algorithms).

Algorithm A(Addition of nonnegative integers): Given nonnegative n-place integers  $(u_1, u_2, \dots, u_n)$  and  $(v_1, v_2, \dots, v_n)$  with radix b, this algorithm forms their sum,  $(w_0, w_1, w_2, \dots, w_n)b$ . (Here  $w_0$  is the "carry", and it will always be equal to 0 or 1).

A1. [Initialise]

Set  $j \leftarrow n$ ,  $k \leftarrow 0$ . (The variable j will run through the various digit positions, and the variable k keeps track of carries at each step.)

A2. [add digits]

Set  $w_j \leftarrow (u_j + v_j + k) \bmod b$ , and  $k \leftarrow (u_j + v_j + k) / b$ . (In other words, k is set to 0 or 1, depending on whether a "carry" occurred or not, i.e. whether  $u_j + v_j + k = b$  or not. At most one carry is possible during the two additions, since we always have

$$u_j + v_j + k \leq (b-1) + (b-1) + 1 < 2b.$$

A3. [Loop on j]

Decrease j by one. Now if  $j > 0$ , go back to step A2;

otherwise set  $w_0 \leftarrow k$  and terminate the algorithm.

There is a possibility of improvement in the above algorithm.

### 3.2 Subtraction

S1: [Initialise]:

Set  $j \leftarrow n$

$k \leftarrow 0$

S2: [Subtract digits.]

Set  $w_i \leftarrow (u_i - v_j + k + b) \bmod b$   
 $k \leftarrow (u_i - v_j + k + b) / b$

S3: [Loop on j]  
 $j \leftarrow j - 1$   
 if  $j > 0$ , go back to step S2  
 else terminate the algorithm

Again the subtraction algorithm has scope for improvement as in the case of addition.

### 3.3 Multiplication

Multiplication is an important building block for public-key cryptosystems. Firstly we consider a simple multiplication which is nothing but the normal pencil and paper method as presented by Knuth. The primitive operation which will help us to perform multiplication is multiplication of one place integer by another one place integer, giving a two place answer.

Algorithm MIM(Multiple Integer Multiplication)

Given: Non negative integer  $u = (u_1, u_2, \dots, u_n)_b$  and  $v = (v_1, v_2, \dots, v_n)_b$ , expressed in radix b, this algorithm forms their radix b product  $(w_1, w_2, \dots, w_{m+n})_b$ .

MIM1. [Initialise]  
 For  $k = m + 1$  to  $m + n$  do  $w_k = 0$ ;  
 $j = m$

MIM2. [Zero multiplier]  
 If  $v_j = 0$  then  $w_j = 0$ ;  
 goto step MIM6.

MIM3. [Initialise]  
 $i = n$ ; carry = 0;

MIM4. [Multiply and Add]  
 $t = u_i * v_j + w_{i+j} + \text{carry}$ .  
 $w_{i+j} = t \text{ mod } b; \text{carry} = t \text{ div } b;$

MIM5. [Loop on i]  
 $i = i - 1;$  if  $i > 0$  then goback to step MIM4 else  $w_i = \text{carry};$

MIM6. [Loop on j]  
 $j = j - 1;$   
 if  $j > 0$  then goback to step MIM2.  
 else stop.

### 3.4 Division

Object: To divide  $(n+m)$  place integer by  $n$ -place integer where  $n$  is very large, of the order of a few hundred decimal digits.

Description: Any division operation essentially consists of dividing a  $(n+1)$ -place integer by an  $n$ -place integer where the divisor is an  $n$  place integer.

A series of such operation constitute a long division of two multiple precision integers. So the problem breaks down to dividing an  $n$  place integer by an  $n$  place integer, i.e.  $u/v$  where  $u=(n+1)$ -place integer and  $v=n$ -place and  $0 \leq u/v < b$  where  $b$  is the base of the integers.

Here  $u = (u_1 u_2 \dots u_n)_b$  and  $v = (v_1 v_2 \dots v_n)_b$ .

The problem is to determine  $q = u/v$ . If we write  $r = u - qv$  then the unique integer satisfying  $0 \leq r < v$ . The algorithm basically attempts to guess the value of  $q$  such that the error is never too large.

Set  $q' = \min((u_0 b + u_1) / v_1, b - 1)$

TH-6846

Algorithm MPID: {Multiple Precision Integer Division}



Given: Non negative integers  $u=(u_1 u_2 \dots u_{m+n})_b$  and  $v=(v_1 v_2 \dots v_n)_b$  expressed in radix  $b$ , where  $v_1 > 0$  and  $n > 1$ , we form the radix- $b$  quotient  $q=(q_1 q_2 \dots q_m)_b$  and the remainder  $r=(r_1 r_2 \dots r_n)_b$ .

MPID1. [Normalize]

$$d = b \text{ div } (v_1 + 1)$$

$$u_0 = 0$$

$$(u_0 u_1 u_2 \dots u_{m+n}) = (u_1 u_2 \dots u_n)_b * d$$

$$(v_1 v_2 \dots v_n) = (v_1 v_2 \dots v_n)_b * d$$

MPID2. [Initialize]  $j=0$

MPID3. [Calculate  $q'$ ]

if  $u_j = v_1$  then  $q' = b - 1$

else  $q' = (u_j b + u_{j+1}) \text{ div } v_1$

while  $v_2 q' > (u_j b + u_{j+1} - q' v_1) b + u_{j+2}$  do

$q' = q' - 1$

MPID4. [Multiply and subtract]

$$(u_j u_{j+1} \dots u_{j+n})_b = (u_j u_{j+1} \dots u_{j+n})_b - q' * (v_1 v_2 \dots v_n)_b$$

MPID5. [Test remainder]

$$q_j = q'$$

if  $(u_j u_{j+1} \dots u_{j+n})_b < 0$  then goto step MPID6

else goto step MPID7

MPID6. [Add back]

$$q_j = q_j - 1$$

$$(u_j u_{j+1} \dots u_{j+n})_b = (u_j u_{j+1} \dots u_{j+n})_b + (0 v_1 v_2 \dots v_n)_b$$

MPID7. [Loop on  $j$ ]

$$j = j + 1$$

if  $j \leq m$  then go back to step MPID3.

MPID8. [Unnormalize]

$$q = (q_0 q_1 q_2 \dots q_m)_b$$

$$r = (u_{m+1} \dots u_{m+n})_b \text{ div } d$$

For ciphering and deciphering we need fast modular reduction operation and fast modular exponentiation. Now we explain the algorithms

for modulo operation and exponentiation. These algorithms can help to compute fast.

### 3.5 Modular reduction:

The standard method for doing the modular exponentiation is by using the well known "repeated square and multiply" technique. The left to right form of this algorithm involves repeatedly squaring and multiplying by a 'local' fixed value (modulo a 'global' fixed value). The 'local' fixed value is dependent on plaintext ( $m$ ), i.e. it is fixed for the duration of the encryption computation. The global fixed value, i.e. the modulus, depends on the key and is therefore fixed for a number of encryption operations. We use this notion of local and global fixed values throughout this section.

This algorithm is concerned with speeding up these modular arithmetic operations. It takes the advantage of local and global fixed values to precompute 'look-up' tables whose use speeds the individual calculations. This also works on a block by block basis, i.e. it makes use of the fact that computers operate on strings of several bits at a time, rather than on individual bits.

In the description of the algorithm we use the following notational convention. We suppose all numbers are stored in 32-bit words, and we denote the individual words of the  $32n$ -bit number  $a$  by

$$a[0], a[1], \dots, a[n-1]$$

where  $a[0]$  contains the least significant word and  $a[n-1]$  the most significant word. Hence

$$a = a[0] + 2^{32}a[1] + 2^{64}a[2] + \dots + 2^{32(n-1)}a[n-1]$$

We then write  $a[]$  for the collection  $a[0], a[1], \dots, a[n-1]$ . We also write  $T$  for  $2^{32}$ . Finally it is kept in mind that the algorithm is described in terms of 8-bit bytes and 32-bit words, it would work equally well with other byte and word lengths.

This algorithm is concerned with modular reduction. Given a  $64n$ -bit number  $k$  and  $32n$ -bit modulus  $d$ , it outputs a  $32n$ -bit number  $k'$ , where

$$k' \equiv k \pmod{d}$$

To do this it uses a table of values pre-computed using the modulus  $d$ . This table does not need to be reevaluated for each computation since it is dependent on a global fixed value. In the context of RSA this algorithm is useful when computing modular squares. Squaring can be made considerably faster than a square-mod method which reduces modulo  $d$  as it works out the answer, so it is quicker to use a fast non-modular squaring algorithm followed by this algorithm.

The idea of the algorithm is to reduce the length of  $k$  by 8 bits (i.e. one byte) at a time. At the beginning of each step it is assumed that  $k$  is  $4n+i+1$  bytes long, (where  $i$  ranges from  $4n-1$  down to  $0$ ), together with an extra bit at the most significant end which may be 1 or 0 (this is left over from previous iteration). At each step of the algorithm, the largest multiple of the  $d$  that can be safely subtracted from  $k$  is subtracted. This process uses the table  $atab$ , set up in advance.

By safely we mean the largest multiple of  $d$  that can be subtracted to leave the result positive, given that only the most significant 9 bits of  $k$  are examined. This results in a value of  $k$  which has the byte under consideration set to either 0 or 1 (this single bit forming the left over bit for the next subtraction).

The table  $atab$  consists of 512 entries of the form

$$atab[i], 0 \leq i \leq 511,$$

where each entry consists of an  $(n+1)$ -word multi-precision integer. In the precomputation phase,  $atab[i]$  is set to the unique integer multiple  $gd$  of  $d$  defined so that

$$\text{int}(gd/T) = i-1 \quad \text{and} \quad \text{int}((g+1)d/T) = i$$

and there is  $\text{int}(x)$  which denotes the unique integer  $s$  satisfying

$$s \leq x < s+1$$

The computation of this table may be achieved by a very simple combination of additions and comparisons; no multiplications or divisions are required.

If one modified the algorithm to reduce by  $w$  bits at a time then  $atab$  would need to contain  $2^{w+1}$  entries.

In the main algorithm,  $atab$  is used to compute a value  $k'$  satisfying

$$k' \equiv k \pmod{d} \quad \text{and} \quad 0 \leq k' \leq T+2d-1$$

The following two steps are repeated  $4n$  times, for a value of  $i$  descending from  $4n-1$  to 0:



(a) Examine bytes  $4n+i+1$  and  $4n+i$ . Byte  $4n+i+1$  will be set to either zero or one, and thus the value  $j$  obtained by regarding the byte pair as an integer will satisfy  $0 \leq j \leq 511$ .

(b) Subtract  $2^{8i}$  times  $atab[j]$  from  $k$ . This will have the effect of clearing byte  $4n+i+1$  and resetting byte  $4n+i$  to either zero or one.

Pseudo-code description:

Input:  $k[]$  (a  $2n$ -word number)

$d[]$  (an odd  $n$ -word constant, where  $d \geq T/2$ .)

Output:  $k[]$  (an  $n$ -word number congruent to the original  $k[]$  modulo  $d$ )

Method: Prior to performing individual computations it is assumed that a  $(512(n+1))$ -word array

$atab[i][j]$  ( $0 \leq i \leq 511, 0 \leq j \leq n$ )

has been set up as follows:

```

atab[0][ ]:=0;
for i:=1 to 511 do
  { Let  $g$  be the unique integer satisfying
     $\text{int}(g.d/T)=i-1$  and  $\text{int}((g+1).d/T)=i$ ;
     $atab[i][ ]:=gd$  }

```

where, as described above,  $atab[i][ ]$  represents the number stored in the  $n+1$  words

$atab[i][0], atab[i][1], \dots, atab[i][n]$

for each  $i$ ,  $g = \text{int}(T.i/d)$ ; the description above is used to emphasise that no divisions are involved in the computation of  $atab$ .

The main algorithm proceeds as follows:

for  $i:=4n-1$  to 0 step -1 do

$\{j:=\text{int}(k[i]/(2^{8i} \cdot T));$

$k[i]:=k[i]-2^{8i} \cdot \text{atab}[j][i]\}$

To complete the process it may be necessary to subtract either  $d$  or  $2d$  from the final  $k$  in order to ensure that it is less than  $T$ , which is sufficient for intermediate results in modular exponentiation. At most one further subtraction of  $d$  will ensure that the result is less than  $d$ .

The two operations in each iteration of the algorithm are very simple. It, although apparently involving a division and an exponentiation, actually merely involves examining the two most significant bytes of  $k$ .

Finally note that during the computation of  $\text{atab}[j][i]$ ,

$d/T \geq 1/2$  and hence  $g < 1022$  for every  $i$

Thus  $g \cdot d$  can be stored in at most  $(n+1)32$ -bit words. This explains why  $\text{atab}[j][i]$  has dimensions allocated to it.

### 3.6 Fast exponentiation:

Here we are describing two methods which are based on same logic but in different ways.

#### a) Square-and-multiply technique

Having seen the basic multiple precision arithmetic routines, we are now in a position to use them for fast modular exponentiation. The aim of this section is to develop faster ways of computing  $x^n \bmod m$ , where both  $m$  and  $n$  are typically very large integers, i.e., around 200 digits each. If we

use a very naive approach and multiply  $x$  with itself  $(n-1)$  times we do get  $x^n$ , but if  $n$  is a 100 digit number then it will take few thousand years to achieve our desired goal even with the fastest of computers. Clearly we have to do much better than this if we wish to have a practical implementation. We first consider a very old method that is still widely used methods for fast exponentiation. For purposes of clarity, we drop the mod notation since it does not play any role in the exponentiation. The idea behind this method, which is known as the "square and multiply method" is very simple: suppose, for example, that we are asked to compute  $x^{16}$ , we could simply start with  $x$  and multiply  $x$  fifteen times as explained above; but it is possible to obtain the same answer in 4 multiplications, if we repeatedly take the square of each partial result, successively  $x^2, x^4, x^8, x^{16}$ .

The same idea applies, in general, to any value of  $n$ , in the following way: We write  $n$  in the binary number system (suppressing zeros at the left). Then replace each "1" by the pair of letters SX, replace each "0" by S, and cross off the "SX" which now appears at the left. The result is a rule for computing  $x^n$ , if "S" is interpreted as the operation of squaring, and if "X" is interpreted as the multiplying by  $x$ . For example, if  $n=23$ , its binary representation is 10111; so we form the sequence SX S SX SX SX and remove the leading SX to obtain the rule SSXSXSX. The rule states that we should "square, square, multiply by  $x$ , square, multiply by  $x$ , square, multiply by  $x$ "; in other words, we should successively compute

$x^2, x^4, x^5, x^{10}, x^{11}, x^{22}, x^{23}$ . If we look a little closely, we will observe that the sequence of computation is the following:

$$x^2, x^2, x^2+1, x^2^3+2, x^2^3+2+1, x^2^4+2^2+2, x^2^4+2^3+2+1 = x(10111)_2.$$

So essentially we are looking at the binary representation of  $n$  from left to right, and if we encounter a "1", we multiply  $x$  to the partial result and square the partial result, else we only square the partial result. This method has the advantage that it brings down the number of computations from  $n$  to about  $\log_2 n$ . However we do have the overhead of converting the exponent from decimal to binary. Since in our case  $n$  is a multi-precision integer, we have to divide by 2 to deduce the binary representation from right to left. We now state the algorithm for exponentiation, based on a right to left scan of the exponent, as described by Knuth.

Algorithm Exp( $x, n$ ) (Right to left binary method for exponentiation):

This algorithm evaluates  $x^n$ , where  $n$  is a positive integer.

A1. [Initialize.]

Set  $N \leftarrow n, Y \leftarrow 1, Z \leftarrow x$ .

A2. [Halve  $N$ .]

(At this point, we have the relation  $x^n = YxZ^N$ .) Set  $N \leftarrow N/2$  and at the same time determine whether  $N$  was even or odd. If  $N$  was even, skip to step A5.

A3. [Multiply  $Y$  by  $Z$ .]      Set  $Y \leftarrow ZxY$ .

A4. [ $N=0$ ?] If  $N=0$ , the algorithm terminates, with  $Y$  as the answer.

A5. [Square  $Z$ .] Set  $Z \leftarrow ZxZ$ , and then return to step A2.

As an example of the working of the algorithm we consider here the steps in the execution of  $x^{23}$ :

	N	Y	Z
After step A1	23	1	x
After step A4	11	x	x
After step A5	11	x	$x^2$
After step A4	5	$x^3$	$x^2$
After step A5	5	$x^3$	$x^4$
After step A4	2	$x^7$	$x^4$
After step A5	2	$x^7$	$x^8$
After step A5	1	$x^7$	$x^{16}$
After step A4	0	$x^{23}$	$x^{16}$

b) SS(I)

We describe a new technique for improving the performance of square-and-multiply exponentiation. Exponentiation of large integers is the basis of several well known cryptographic algorithms such as RSA algorithm. The exponentiation problem is to compute  $m^e \pmod{N}$ , where  $m$  and  $N$  are very large integers, and  $e$  is the integer exponentiation, such that  $e$  has binary representation ' $e_1e_2\dots e_{n-1}e_n$ ' where  $e_1$  is the most significant bit. We also treat  $e$  as a bit string of length  $n$  denoted as  $e[1,n]$ , where  $n$  is typically 512 or 1024. Throughout this section, for simplicity  $m^j \pmod{N}$  is written as  $m^j$  for any integer  $j$ .

Various representations of  $e$  have been suggested with the same goal of reducing the number of multiplications involved. The best published approach to date is the 'modified signed digit' representation which requires an average of  $n/3$  multiplications at the cost of precomputing  $m^j$ . Recently, a further improvement, based on a 'string replacement' representation, was reported by Mitchell, who proposed a family of

algorithms k-SR where k is an integer parameter that determines the number of exponents of m needed to be precomputed. Mitchell proved that 2-SR and 5-SR reduce the average number of multiplications to  $n/3$  and  $8n/31$ , respectively. In fact, this number is reduced to  $n/4$  when  $k'$  .

Analysis of our algorithm SS(l): On n-bit exponents has shown that the expected number of multiplications required tends to  $n/(l+1)$  for large n. Here we only compare the performance of our technique with that of the approach of Mitchell because it is the best reported algorithm to date.

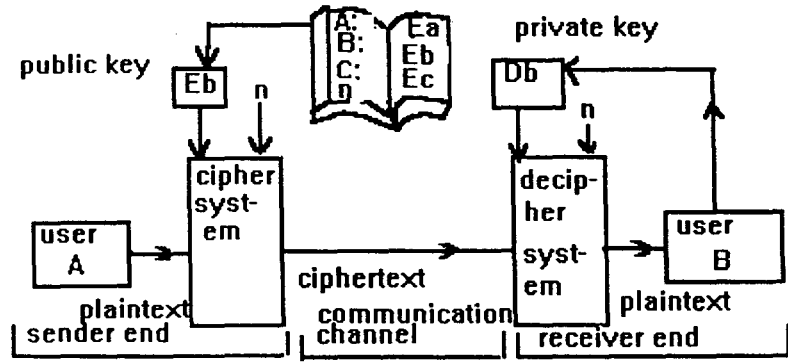
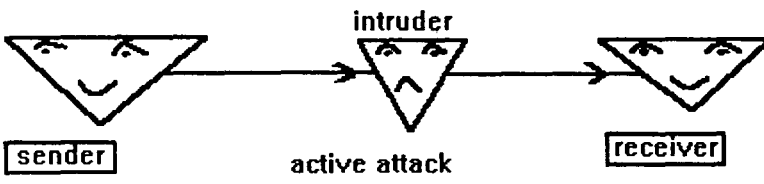
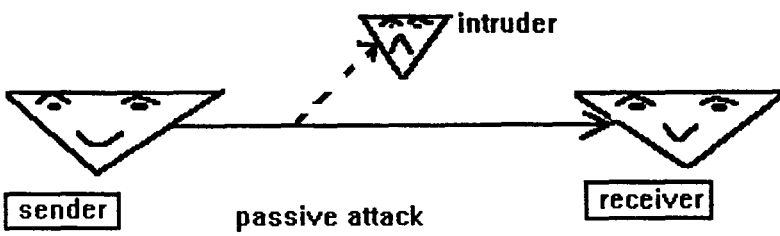
Algorithm: It is a family of algorithm that are used to compute  $m^e$ . Each algorithm is parameterised by the value l, the maximum length of the exponent to be precomputed.

The idea is to precompute a set of exponents of m:  $m^{2^{i+1}} \forall i$  belongs to  $[1...(2^{l-1}-1)]$ , i.e. the set of all odd integers between 3 and  $2^{l-1}$ . The precomputed exponents are stored in a table  $H[1...(2^{l-1}-1)]$  where  $H[i]$  contains  $m^{2^{i+1}}$ . The algorithm reduces the number of multiplications through this exponent table. It scans  $e[1..n]$  starting from  $e_1$ . The variable x always stores  $m^{e[1..i]}$  for some i belongs to  $[1..n]$ . If  $e[i+1]$  is a '0', x becomes  $x^2$ , i.e.  $m^{e[1..i+1]}$ . Otherwise, i.e. if  $e[i+1]$  is a '1', the algorithm first recognises the longest bit pattern  $e[(i+1)..j]$  such that  $m^{e[(i+1)..j]}$  is in H, secondly updates x by repeatedly self-squaring it j-i times, and then thirdly

multiplies  $x$  by the precomputed value  $m^{e[(i+1)..j]}$ . The SS(l) algorithm is as

follows:

```
x ← 1; i ← 0;
Loop { /* the invariant is: x = m^{e[1..i]} */
  while(i < n) and (e_{i+1} ≠ '0')
    { i ← i + 1; x ← x^2(mod N); }
  if(i ≥ n) then done and return x as answer.
  j ← i + 1;
  while(j > n) or (e_j = '0') { j ← j - 1; }
  a = the value e[(i+1)..j]; b ← (a-1)/2;
  while(i < j) { i ← i + 1; x ← x^2(mod N); }
  x ← x * H[b](mod N);
} /* End Loop */
```





## CHAPTER 4

### DESIGN AND IMPLEMENTATION

In RSA cryptosystem, at the sender end our system will convert alphabetic plaintext into integer form of plaintext, compute modular exponentiation of integer plaintext and at the receiver end it recovers the plaintext from ciphertext and reformat it into alphabetic plaintext.

For storing large integer I can allocate memory dynamically or statically. Dynamic allocation is used where we do not know upper bound. In RSA algorithm we know the upper bound i.e. multiplication of two large prime numbers chosen has the maximum fixed length. 32 words of unsigned integer is taken as an array to store a large integer. I have used word size 16 for every large number so that multiplication of two such large numbers can be accommodated in 32 word array of integers as in the case of casting. There is no in-built multiprecision arithmetic in our system so we need to develop functions for multiprecision arithmetic for performing various operations, i.e. modulo arithmetic and exponentiation.

Casting is required where we multiply two 16 bit numbers because without casting, the result will be stored in 16 bit which may cause overflow. In various places where in an arithmetic expression 16 bit subtraction is taking place and addition of 17 bit number (stored in 32 bit number) is placed afterward, then there may happen subtraction of big 16 bit number from small 16 bit number resulting 2's complement of result and added to 17 bit number gives carry while adding small 16 bit number

into 17 bit number and subtraction of big 16 bit number from it will not produce any carry which is required, so we add a 16 bit number in 17 bit number then we subtract big 16 bit number from it to get correct carry and result.

To avoid overflow we use casting again where addition of two or more 16 bit numbers, is performed. Coercion is taking place where we are operating any 16 bit number into 32 bit number, in these places in expression 32 bit number must come first and others afterwards.

For computing multiplicative inverse I used Euclid's algorithm but we have not represented large -ve numbers in our large number representation. Computed  $y_i$ 's are alternatively -ve and +ve.  $y_i$  is -ve if  $i$  is odd otherwise +ve. We have used a flag which indicate sign of the number. When result will be positive then it is our required result otherwise we have to subtract the number without sign from modulo number to get the required result.

In multi-precision division where divisor is one word in length, we use our pencil and paper technique to get the result. In multi-precision addition, subtraction, multiplication, and division, index of represented positional words of most significant is one and index of least significant word is  $n$ . but I designed functions for these operations so that less significant word's index is small integer and more significant word's index is big integer. I have designed different algorithm for different type of operation e.g. logical operations, assignment operations, input ,output etc.

**Implementation:** I have developed this software in the environment of Turbo C under DOS, and the hardware platform is Pentium(133 Hz). Very few changes can make it able to be run in ANSI C under unix. It can be developed in any version of C, i.e. C++, VC++ etc. It can be made portable, doing some changes like using preprocessing statements, that will generate code for the environment used at the time of compilation. I did some programming level code optimization. We can do some more programming level code optimization by doing walk through the program and keeping it in mind that we have to optimize our code. We can make it to take input either from standard input or from file and to give output to standard output, communication channel or files. We can also make it so general so that same program have options to take input from standard input, files or communication channels at the sender end and to give output to standard output, communication channel or file at the receiver end. I have implemented it on the basis of block ciphering mechanism. I can make it on the basis of stream ciphering mechanism or give option for that. At last I want to say that it is not absolute general software but I can make it absolute to handle any mode, any type in input and output systems.

Further after designing very few steps we can develop RSA based public key cryptosystem and also any other public key cryptosystem based on the operations of modular exponentiation, modulo operations, modulo multiplications, multiplicative inverse etc.

## CHAPTER 5

### CODING & TESTING

Since our design was in more detailed manner so that our coding is done more mechanically. It does not mean that coding is absolute mechanical because I have some limitations while designing. In designing process we have supposed that a function will take some defined type of input and give some specific type of output. To achieve this output from input I have designed algorithms and in coding step all the steps of algorithms must be syntactically, semantically and logically correct so that it must be compiled without any compilation and linking error. When it is linked then I have tested it giving some test data which can execute every step of the function and verified its output from my result data computed manually. When there were some logical errors that gave incorrect output then I have corrected it so that it gave correct output and all the steps of function are executed giving correct output. I have done coding this type and tested for every function. Yet I have been strucked in some functions which are calling another tested function but needed correction in tested function to execute and give output properly. At last all functions were logically correct, while there are several functions those are dependent on each other, and gave logically correct output.

Program for this software comprises of approximately nine hundred lines of code excuding the files included.

Precomputation of table takes 0.329670 second. Encryption of 'A' takes 0.219780 second, and encrypted text of 'A' takes 0.549451 second for decryption.

Table given below shows that what length of text takes how much time for encryption and decryption.

<b>Text length ( in characters)</b>	<b>Time for encryption ( in seconds )</b>	<b>Time for decryprion ( in seconds )</b>
1	1.483516	4.505495
2	1.483516	4.505495
3	1.483516	4.505495
4	1.483516	4.505495
5	1.483516	4.505495
6	1.593407	4.505495
7	1.593407	4.505495
8	1.593407	4.505495
9	1.593407	4.505495
10	1.593407	4.505495
11	1.593407	4.505495
12	1.593407	4.505495
13	1.593407	4.505495
14	1.648352	4.505495
15	1.648352	4.505495
16	1.648352	4.505495
17	1.648352	4.505495
18	1.648352	4.505495
19	1.648352	4.505495
20	1.648352	4.505495
22	1.648352	4.505495
22	1.703297	4.505495
23	1.703297	4.505495
24	1.703297	4.505495
25	1.703297	4.505495
26	1.703297	4.505495
27	1.703297	4.505495
28	1.703297	4.505495
29	1.703297	4.505495
30	1.703297	4.505495
31	1.703297	4.505495
32	1.703297	4.505495

## CHAPTER 6

# COMPARISON

We compare RSA cryptosystem with the RSA based cryptosystem.

RSA cryptosystem:

In RSA cryptosystem we have to perform various operations to generate keys and then use fast exponentiation for ciphering and deciphering. We need multiplicative inverse at the time of generation of keys. At the time of application we need only modular exponentiation. The time taken by this cryptosystem may be little bit more than RSA based cryptosystem but in it there are no complex computations and complexity of it is also less compare to RSA based cryptosystem. From the point of security it is more secure than RSA based cryptosystem.

RSA based cryptosystem:

In RSA based cryptosystem we need all operations of RSA cryptosystem for generating keys, then using these generated keys we need only multiplicative inverse, addition, subtraction and modulo operation to cipher and decipher. It is less secure than RSA cryptosystem, it is more complex than RSA cryptosystem but it is faster than RSA cryptosystem.

## APPENDIX A- Source code

```
add_large_integer(Number a[],Number b[],Number c[]);    /*c=a+b*/
subtract_large_int(Number a[],Number b[],Number c[]);    /*c=a-b*/
multiply_large_int(Number u[],Number v[],Number w[]); /*w=u*v */
devide_large_integer(Number u[],Number v[],Number q[],Number r[]);
                    /* u=q*v + r */
print_blarge_integer(Number a[]);
sub(Number b[],int num);
assign(Number a[],Number b[]);
create_table(Number a[],Number d[],int l);
modular_reduction(Number a[],Number k[],Number d[], int l);
create_h(Number a[],Number h[], Number m[],Number d[],int l);
mod_exp_ssl(Number a[],Number m[],Number e[],d[],x[],int l);
gcd(Number u[],Number v[], Number a[]);
mult_inv_mod(Number a[],Number b[],Number y1[]);/* b*y1=1mod(a) */
make_rand_number(Number x[],int l);
assign_large_integer(Number b[],char *p);
scan_text_and_make_to_integer(Number g[]);
make_integer_to_text_and_print(Number g[]);
```

```

main()
{
    int i,j,q,k,l;
    Number t[ARRAYSIZE],g[ARRAYSIZE],r[ARRAYSIZE],n[ARRAYSIZE];
    Number h[512][ARRAYSIZE];
    Number atab[512][ARRAYSIZE];
    Number a[ARRAYSIZE],b[ARRAYSIZE],c[ARRAYSIZE],d[ARRAYSIZE];
    char *p,*p1="170141183460469231731687303715884105727"
        ,*p2="680564733841876926926749214863536422887";

    fflush(stdin);
    assign_large_integer(a,p1);
    assign_large_integer(b,p2);
    multiply_large_integer(a,b,n);
    sub(a,1);
    sub(b,1);
    multiply_large_integer(a,b,c);
    make_rand_number(a,7);
    while(1)
    {
        gcd(c,a,b);
        devide_large_integer(a,b,g,r);
        assign(a,g);
        if(equal_to_one(b)) break;
    }
    mult_inv_mod(c,a,t);
    l=32;
    while(n[--l]==0);
    l++;
    create_table(atab,n,l);
    scan_text_and_make_to_integer(g);
    mod_exp_ssl(atab,g,a,n,r,l);
    mod_exp_ssl(atab,r,t,n,g,l);
    make_integer_to_text_and_print(g);
}

add_large_integer(Number a[],Number b[], Number c[]) /* c=a+b*/
{
    const Lnumber d=BBASE;
    int i,n=ARRAYSIZE, carry=0;
    Lnumber t;
    while((a[--n]==0) && (b[n]==0) && (n>=0) );
    for(i=0;i<n+1;i++)
    {

```



```

    t=(Lnumber)a[i]+b[i]+carry;
    c[i]=t%d;
    carry=t/d;
}
c[i++]=carry;
while(i<ARRAYSIZE) c[i++]=0;
}

subtract_large_integer(Number a[],Number b[],Number c[])
{
    const Lnumber d=BBASE;
    Lnumber t;
    int i,neg=0,n=ARRAYSIZE, carry=1;
    if(greater_than(b,a)) neg=1;
    while((a[--n]==0) && (b[n]==0) && (n>=0));
    for(i=0;i<n+1;i++)
    {
        t=d+a[i]-b[i]+carry-1;
        c[i]=t%d;
        carry=t/d;
    }
    if(neg) c[i]=-1;
    while(i<ARRAYSIZE) c[i++]=0;
}

multiply_large_integer(Number u[],Number v[],Number w[])
{
    Lnumber t,c,d=BBASE;
    int i,j,k,m,n;
    m=ARRAYSIZE;n=ARRAYSIZE;
    make_zero(w);
    while((u[--n]==0) && (n>=0));
    while((v[--m]==0) && (m>=0));
    for(j=0;j<m+1;j++)
    {
        if(v[j]!=0)
        {
            c=0;
            for(i=0;i<n+1;i++)
            {
                t=(unsigned long int)u[i]*v[j]+w[i+j]+c;
                w[i+j]=t%d;
                c=t/d;
            }
        }
    }
}

```

```

        w[i+j]=c;
    }
}
}

```

```

divide_large_integer(Number u[],Number v[],Number q[],Number r[])
{
    Number q1,q2,t[ARRAYSIZE],s[ARRAYSIZE],p[ARRAYSIZE],
        o[ARRAYSIZE],x[ARRAYSIZE];
    Lnumber ck1,ck2,ck3,b=BBASE;
    int i,j,m,n;
    m=n=ARRAYSIZE;
    for(i=0;i<m;i++) { t[i]=0; q[i]=0; r[i]=0; o[i]=0;}
    while((u[--m]==0) && (m>=0));
    while((v[--n]==0) && (n>=0));
    if(n<0){ fprintf(stderr,"\ndevide by zero error"); return;}
    if(n==0)
    {
        assign(q,u);
    }
    else
    {
        q2=t[0]=b/((Lnumber)v[n]+1);
        multiply_large_integer(u,t,r);
        multiply_large_integer(v,t,s);
        for(i=0;i<m-n+1;i++)
        {
            if(r[m+1-i]==s[n]) q1=b-1;
            else
                q1=((Lnumber)r[m+1-i]*b+r[m-i])/s[n];
            ck1=(Lnumber)r[m+1-i]*b+r[m-i] -(Lnumber)q1*s[n];
            ck2=(Lnumber)s[n-1]*q1;
            ck3= ck1*b+r[m-i-1];
            while( (ck1<b) && (ck2>ck3) )
            {
                q1--;
                ck1=(Lnumber)r[m+1-i]*b+r[m-i] -(Lnumber)q1*s[n];
                ck2=(Lnumber)s[n-1]*q1;
                ck3= ck1*b+r[m-i-1];
            }
            t[0]=q1;
            multiply_large_integer(s,t,p);
            for(j=m-i-n;j<m+2-i;j++) o[i+j-m+n]=r[j];
            q[m-i-n]=q1;
            if(greater_than(p,o))

```

```

    {
        q[m-i-n]--;
        subtract_large_integer(p,s,x);
        subtract_large_integer(o,x,p);
        for(j=m-i-n;j<m+2-i;j++) r[j]=p[i+j-m+n];
    }
    else
    {
        subtract_large_integer(o,p,x);
        for(j=m-i-n;j<m+2-i;j++) r[j]=x[i+j-m+n];
    }
}
}
}

```

```

create_table(Number a[][ARRAYSIZE], Number d[],int l)

```

```

{
    Number atab[512][ARRAYSIZE];
    int i,n; Number t[ARRAYSIZE],g[ARRAYSIZE],r[ARRAYSIZE];
    for(i=0;i<ARRAYSIZE;i++) { a[0][i]=0; t[i]=0; }
    for(i=1;i<512;i++)
    {
        t[i]=i;
        devide_large_integer(t,d,g,r);
        multiply_large_integer(d,g,a[i]);
    }
}

```

```

modular_reduction(Number a[][ARRAYSIZE],Number k[],Number d[], int
l)

```

```

{
    Number j,n,t[ARRAYSIZE],q[ARRAYSIZE],r[ARRAYSIZE];
    int i;
    for(i=2*l-1;i>=0;i--)
    {
        j=k[l+i/2];
        if(i%2) j=j/256;
        assign(t,a[j]);
        left_shift_by_n(t,i/2);
        if(i%2) mul(t,256);
        subtract_large_integer(k,t,k);
    }
    while(greater_than(k,d))
    {
        print_blarge_integer(k);
    }
}

```

```

    print_blarge_integer(d);
    devide_large_integer(k,d,q,r);
    assign(k,r);
}
}

create_h(Number a[],Number h[], Number m[],Number d[],int l)
{
    int i;
    multiply_large_integer(m,m,h[0]);
    modular_reduction(a,h[0],d,l);
    multiply_large_integer(m,h[0],h[1]);
    modular_reduction(a,h[1],d,l);
    for(i=2;i<8;i++)
    {
        multiply_large_integer(h[0],h[i-1],h[i]);
        print_blarge_integer(d);
        modular_reduction(a,h[i],d,l);
        print_blarge_integer(d);
    }
}

mod_exp_ssl(Number at[],m[],Number f[],Number d[],x[],int l)
{
    int i,j,n,a,k,b,e1=10;
    char *e;
    Number h[512][ARRAYSIZE],y[ARRAYSIZE];
    create_h(at,h,m,d,l);
    n=strlen(e);
    n--;
    for(i=0;i<ARRAYSIZE;i++) x[i]=0;
    x[0]=1;
    i=-1;
    while(1)
    {
        while( (i<n) && (e[i+1]!='0') )
        {
            i++;
            multiply_large_integer(x,x,y);
            modular_reduction(at,y,d,l);
            assign(x,y);
        }
        if(i>=n) return;
        j=i+e1;
        while( (j>n) || (e[j]!='0') ) j--;
    }
}

```

```

a=0;
for(k=i+1;k<j+1;k++)
{
    a=a*2;
    if(e[k]=='1') a++;
}
b=(a-1)/2;
while(i<j)
{
    i++;
    multiply_large_integer(x,x,y);
    modular_reduction(at,y,d,l);
    assign(x,y);
}
multiply_large_integer(h[b],x,y);
modular_reduction(at,y,d,l);
assign(x,y);
}
}

```

```

gcd(Number u[],Number v[], Number a[])
{
    int i;
    Number q[ARRAYSIZE],r[ARRAYSIZE],b[ARRAYSIZE];
    assign(a,u);
    assign(b,v);
    if(greater_than(b,a)) swap (a,b);
    do{
        devide_large_integer(a,b,q,r);
        assign(a,b);
        assign(b,r);
    } while( not_zero(r) );
}

```

```

mult_inv_mod(Number a[],Number b[],Number y3[])
{
    Number
zero[ARRAYSIZE],o[ARRAYSIZE],r1[ARRAYSIZE],r2[ARRAYSIZE],
y2[ARRAYSIZE],q[ARRAYSIZE],r3[ARRAYSIZE],y1[ARRAYSIZE];
    int flag=0;
    assign(r1,a);
    assign(r2,b);
    make_zero(zero);
    make_zero(y1);
    make_zero(y2);
}

```

```

y2[0]=1;
do
{
    devide_large_integer(r1,r2,q,r3);
    multiply_large_integer(y2,q,o);
    add_large_integer(y1,o,y3);
    assign(r1,r2);
    assign(r2,r3);
    assign(y1,y2);
    assign(y2,y3);
    flag++;
} while( !equal_to_one(r2) );
if(flag%2){ subtract_large_integer(a,y2,y3);}
}

```

```

sub(Number b[],int num)
{
    const Lnumber d=BBASE;
    Lnumber t;
    int c=num,i,n=ARRAYSIZE,carry=1;
    while((b[--n]==0) && (n>=0));
    for(i=0;i<n+1;i++)
    {
        t=d+b[i]+carry-c-1;
        b[i]=t%d;
        carry=t/d;
        c=0;
        if(carry==1) return;
    }
    if(carry==0) b[i]=-1;
}

```

```

assign(Number a[],Number b[])
{
    int i,m=ARRAYSIZE;
    while((b[--m]==0) && (m>=0)) a[m]=0;
    for(i=0;i<m+1;i++) a[i]=b[i];
}

```

```

make_rand_number(Number x[],int l)
{
    int i;
    make_zero(x);
    for(i=0;i<l;i++)
        x[i]=rand();
}

```

```

}

assign_large_integer(Number c[],char *str)
{
    char *p,b[DIGIT];
    Number a[ARRAYSIZE];
    int len,i,j,k,l,n;
    p=str;
    len=strlen(str);
    n=0;
    j=len/DIGIT;
    k=len%DIGIT;
    for(l=0;l<j;l++)
    {
        for(i=0;i<DIGIT;i++) b[i]=p[len-DIGIT+i];
        a[n]=atol(b);
        n++;
        len -= DIGIT;
    }
    if(k)
    {
        for(i=0;i<k;i++) b[i]=p[len-k+i];
        b[i]='\n';
        a[n]=atol(b);
        n++;
    }
    for(i=n;i<ARRAYSIZE;i++) a[i]=0;
    large_dinteger_to_large_binteger(a,c);
}

print_blarge_integer(Number a[])
{
    int i,j,c,n=ARRAYSIZE;
    Number b[ARRAYSIZE],t;
    for(i=0;i<ARRAYSIZE;i++) b[i]=0;
    while((a[--n]!=0)&&(n>=0));
    for(i=0;i<n+1;i++)
    {
        t=32768;
        for(j=0;j<BITSINWORD;j++)
        {
            mul_d(b,2); if( a[n-i] & t ) add_d(b,1);
            t>>=1;
        }
    }
}

```

```

n=ARRAYSIZE;
while((b[--n]==0)&&(n>=0));
printf("\nBlarge integer:\t");
if(n>=0) printf("%d",b[n]);
else printf("%d",b[n+1]);
for(i=1;i<n+1;i++) printf("%04d",b[n-i]);
}

scan_text_and_make_to_integer(Number g[])
{
int i,n,j;
char c,p[38],q[76],a[3];
printf("\nEnter number of characters(<39)\t");
scanf("%d",&n);
printf("\nEnter text\n");
for(i=0;i<n;i++)
{
c=getche();
if((c==' ')||(c=='\t')) j=0;
else
j=toupper(c)-'A'+1;
q[2*i]=j/10+48;
q[2*i+1]=j%10+48;
}
q[2*i]=0;
printf("\n");
assign_large_integer(g,q);
}

make_integer_to_text_and_print(Number a[])
{
Number t,b[ARRAYSIZE];
int i,n=32,j,k,l;
char q[38];
make_zero(b);
while((a[--n]==0)&&(n>=0));
for(i=0;i<n+1;i++)
{
t=32768;
for(j=0;j<BITSINWORD;j++)
{
mul_d(b,2);
if( a[n-i] & t ) add_d(b,1);
t>>=1;
}
}
}

```



```
}
n=32;
while((b[--n]==0) && (n>=0));
for(i=n;i>=0;i--)
{
    l=2*(n-i);
    j=b[i]/100;
    k=b[i]%100;
    q[l]=j?'A'+j-1:' ';
    q[l+1]=k?'A'+k-1:' ';
}
q[l+2]=0;
printf("\n%s",q); }
```

## APPENDIX B- Test data & its output

large integer:

11579208923731619542357098500868790784833589034528695631923  
865220015249057384 9 /\* multiplication of two primes \*/

large integer:

697014061789277386526146338816173 /\* public key \*/

large integer:

56885506409786238714224180125748840116766189052022173646161  
77785929307677929 3 /\* private key \*/

/\* Cipherring and Decipherring starts from here \*/

1. Enter number of characters(<33) 28

Enter text: jawahar lal nehru university

large integer:

36274927342744003827380040313444470047403548314445354651

large integer:

15268634331665784121278047287744126943682032611422621035720  
855585189302321280

large integer:

36274927342744003827380040313444470047403548314445354651  
jawahar lal nehru university

2. Enter number of characters(<33) 28

Enter text: Jawahar Lal Nehru University

large integer:

10274927342744001227380014313444470021403548314445354651

large integer:

62208948520097550668833716455927057041814317811576300194080  
969765223077815352

large integer:

10274927342744001227380014313444470021403548314445354651  
Jawahar Lal Nehru University

3. Enter number of characters(<33) 32

Enter text: Jawahar Lal Nehru was 1st PM of

large integer:

10274927342744001227380014313444470049274500094546001613004  
13200

large integer:

23152643646199069923414494709173866688528411866360193881120  
695081267097095275

large integer:

10274927342744001227380014313444470049274500094546001613004  
13200

Jawahar Lal Nehru was 1st PM of

4. Enter number of characters(<33) 1

Enter text: a

large integer:

27

large integer:

64508007307897932527864393722951027213960417910581969560210  
017207690791050877

large integer:

27

a

5. Enter number of characters(<33) 1

Enter text: A

large integer:

1

large integer:

1

large integer:

1

A

6. Enter number of characters(<33) 15

Enter text: Hello Mr Brahma

large integer:

83138384100134400024427343927

large integer:

77767943450200751020937238582056540623209359832601346289647  
317405672807003871

large integer:

83138384100134400024427343927

Hello Mr Brahma

## APPENDIX C- Bibliography

1. Albrecht & Beutelspacher; Cryptography; Mathematical Association of America, 1994
2. Gollmann D., Han Y., Mitchell C.J.; Redundant Integer Representations and Fast Exponentiation; Kluwer Academic publishers, 1996
3. Introduction to EDI- A primer//The evolution of electronic commerce; Vide: [http://www.geis.com/geis/edi/edifaq\\*.html](http://www.geis.com/geis/edi/edifaq*.html)
4. Knuth D.E.; The art of computer programming: Semi numerical algorithms; Addison-Wesley Publishing company; vol 2, 2nd Edition, 1981
5. Konheim A.G.; Cryptography: A primer; Mathematical Science Deptt., IBM Thomas J. Watson Research Centre.
6. Lam K.Y. & Hui L.C.K.; Fast square-and-multiply exponentiation for RSA; Electronics letters, vol. 30, No. 17, 18th Aug 1994
7. Meyer C.H. & Matyas S.M.; Cryptography: A new dimension in computer data security; John Wiley & Sons, 1982
8. Mitchell C.J.; Another improvement to square-and-multiply exponentiation; Technical report CSD-TR-93-21A (Revised version), Royal Holloway University of London, Department of computer science, Egham, Surrey TW20 0EX, England, 31 Jan 1994
9. Pai T.G. & Prasad U.C.; C implementation of multi-precision arithmetic algorithms for public key cryptosystems; IIT Delhi, April 1997
10. Roy S.A.; Banking & Electronic commerce; CSI communication, June 1997
11. Sandhu R. & Samarati P.; Authentication, Access control and Audit; ACM computing survey, vol. 28, No. 1, March 1996
12. Selby & Mitchell C.; Algorithms for software implementations of RSA//IEEE proceedings, vol. 136 pt. E, No. -3, May 1989

13. Tanenbaum A.S.; Computer networks, 3rd edition; P.H.I. Pvt. Ltd., N. Delhi-110001, 1997

14. Venkaiah V.Ch.; An RSA based public key cryptosystem for secure communication; Proc. Indian Academic Science (Math. Sci.), vol 102, No. 2, August 1992