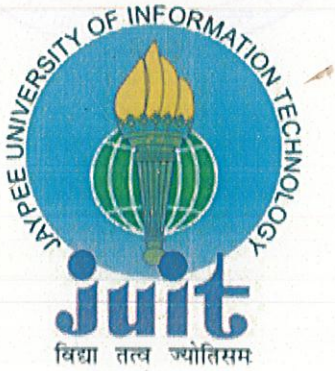
IMPLEMENTATION OF PUBLIC KEY CRYPTOGRAPHY SYSTEM

Project Report submitted in partial fulfillment of the requirement for the degree of
Bachelor of Technology.
in
Electronics and Communication Engineering
under the Supervision of
Dr. T.S. Lamba
by

Geetika Sharma(081034)
Medha Parashar(081056)
Nukul Sehgal(081312)

to



Jaypee University of Information and Technology
Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that the work entitled **"Implementation of Public Key Cryptography System"** submitted by **"Geetika Sharma(081034), Medha Parashar(081056) and Nukul Sehgal (081312)"** in partial fulfillment for the award of degree of Bachelor of Technology in Electronics and Communication Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date: 29/5/2012

TSLamba

Prof. T.S. Lamba
Dean (A&R)
Department of ECE

Acknowledgement

We acknowledge with gratitude to our supervisor **Prof. T.S. Lamba** for his continuous guidance and encouragement throughout the whole study period which helped us in completing the project work, in time.

We would also like to thank **Mr. Mohan Sharma**, In-charge of Project Lab, for all his valuable assistance in the project work and for his timely cooperation in the conduct of our project work.

Finally, yet importantly, we would like to express our heartfelt thanks to our beloved parents for their blessings, our friends and classmates for their help and wishes for the successful completion of this project.

Date:

Geetika Sharma
Medha Parashar
Nukul Sehgal

Table of Content

List of Figures

Abstract

This thesis presents the implementation of the RSA algorithm, which is one of the most widely used Public Key Cryptosystems (PKC) in the world. In RSA Cryptosystem, modular exponentiation of large integers is used for both encryption and decryption processes. The security of the RSA increases as the number of the bits increase. Here we have implemented RSA of 512 bits.

The implementation of RSA is completed in the following steps.

1. Large number arithmetic
2. Primality testing
3. Random Number Generation
4. Encryption and Decryption

Representing truly enormous integers requires stringing digits together.

• Arrays of Digits — The easiest representation for long integers is as an array of digits, where the initial element of the array represents the least significant digit. Maintaining a counter with the length of the number in digits can aid efficiency by minimizing operations which don't affect the outcome.

In this section, we will implement the major arithmetic operations for the array-of-digits representation. Dynamic memory allocation and linked lists provide an illusion of being able to get unlimited amounts of memory on demand. However, linked structures can be wasteful of memory, since part of each node consists of links to other nodes. What dynamic memory really provides is the freedom to use space where you need it.

If you wanted to create a large array of high-precision integers, a few of which were large and most of which were small, then you would be far better off with a list-of-digits representation, since you can't afford to allocate enormous amounts of space for all of them. The large number arithmetic thus makes it easy to perform operations like addition, subtraction, multiplication, division and exponentiation on large numbers.

Using the library created, large random numbers are generated which are tested for primality. The Miller–Rabin test is implemented for primality testing as it is strictly stronger than primality tests. If n is composite then the Miller–Rabin primality test declares n probably prime with a probability at most 4^{-k}. On average the probability that a composite number is declared *probably prime* is significantly smaller than 4^{-k}.

Further more, the generation of public and private keys for the purpose of encryption and decryption includes modular exponentiation. The data is then encrypted by the sender using the public-key of the receiver which is freely available and on the receiver end, data is decrypted using the private key of the receiver which is kept secret by the receiver.

CHAPTER 1: CRYPTOGRAPHY

Cryptography is defined as "the science and study of secret writing," it concerns the ways in which communications and data can be encoded to prevent disclosure of their contents through eavesdropping or message interception, using codes , ciphers , and other methods, so that only certain people can see the real message. Although the science of cryptography is very old, the desktop-computer revolution has made it possible for cryptographic techniques to become widely used and accessible to non-experts.

1.1 The Need for Cryptography

This is an era of electronic connectivity. The explosive growth in computer system has increased the dependence of both organizations and individuals on information stored and communicated using these systems. With this there is a growing need to protect these network based communication and data stored in computers from electronic eavesdropping, electronic fraud, hackers and viruses. Therefore, maintenance of security has become a prime concern in the field of communication. Cryptography is the science that provides this required security. Cryptography provides following services:

a) **Confidentiality:** assuring that private data remains private.
b) **Authentication:** assuring the identity of all parties attempting access.
c) **Authorization:** assuring that a certain party attempting to perform a function has the permissions to do so.
d) **Data Integrity:** assuring that an object is not altered illegally.
e) **Non-Repudiation:** assuring against a party denying a data or communication that was initiated by them.

The ability to protect and secure information is vital to the growth of electronic commerce and to the growth of the Internet itself. Many people need or want to use communications and data security in different areas. Banks use encryption methods all around the world to process financial transactions. These involve transfer of huge amount of money from one bank to another. Banks also use encryption methods to protect their customers ID numbers at bank automated teller machines.

"As the economy continues to move away from cash transactions towards 'digital cash', both customers and merchants will need the authentication provided by unforgeable digital signatures in order to prevent forgery and transact with confidence."

This is an important issue related to the Internet users. There are many companies and even shopping malls selling anything from flowers to bottles of wines over the

Internet and these transactions are made by the use of credit cards and secure Internet browsers including encryption techniques. The customers over the Internet would like to be secure about sending their credit card information and other financial details related to them over a multi-national environment. It will only work by the use of strong and unforgettable encryption methods.

Also business and commercial companies with trade secrets use or would like to use encryption against high-tech eavesdropping and industrial espionage. Professionals such as lawyers, doctors, dentists or accountants who have confidential information throughout their activities will need encryption if they will rely on the use of Internet in the future. Criminals do use encryption and will use it to cover their illegal activities and to make untraceable perfect crimes possible. More important, people need or desire electronic security from government intrusions or surveillance into their activities on the Internet.

These days Cryptography is involved in various day to day activities .It plays its role in :

a) Internet
b) e-commerce
c) Smart cards
d) Credit cards
e) ATM
f) ID cards and health insurance cards
g) Wireless communication and various other spheres of communication.

So we have chosen to explore and work in this inevitable field of communication.

1.2 History:

Cryptography (or *cryptology*; from Greek, "hidden, secret"; and *graphein*, "writing", or "study", respectively) is the practice and study of techniques for secure communication in the presence of third parties (called adversaries). More generally, it is about constructing and analyzing protocols that overcome the influence of adversaries and which are related to various aspects in information security such as data confidentiality , data integrity, and authentication. Modern cryptography intersects the disciplines of mathematics, computer science , and electronic engrneering.

Cryptography prior to the modern age was effectively synonymous with *encryption*, the conversion of information from a readable state to apparent nonsense. The originator of an encrypted message shared the decoding technique needed to recover the original information only with intended recipients, thereby precluding

unwanted persons to do the same. Since World War I and the advent of the computer, the methods used to carry out cryptology have become increasingly complex and its application more widespread.

Modern cryptography is heavily based on mathematical theory and computer science practice; cryptographic algorithms are designed around computational hardness assumptions, making such algorithms hard to break in practice by any adversary. It is theoretically possible to break such a system but it is infeasible to do so by any known practical means. These schemes are therefore termed computationally secure; theoretical advances (e.g., improvements in integer factorization algorithms) and faster computing technology require these solutions to be continually adapted.

There exist information-theoretically secure schemes that provably cannot be broken even with unlimited computing power—an example is the one-time pad—but these schemes are more difficult to implement than the best theoretically breakable but computationally secure mechanisms.

1.3 Types of Cryptography:

Currently, most cryptography used in practice is key based, that is a string of bits, that is used to encode the clear text into cipher text and back again to clear text when required. Two types of key based cryptography exist, based on the availability of the key publicly:

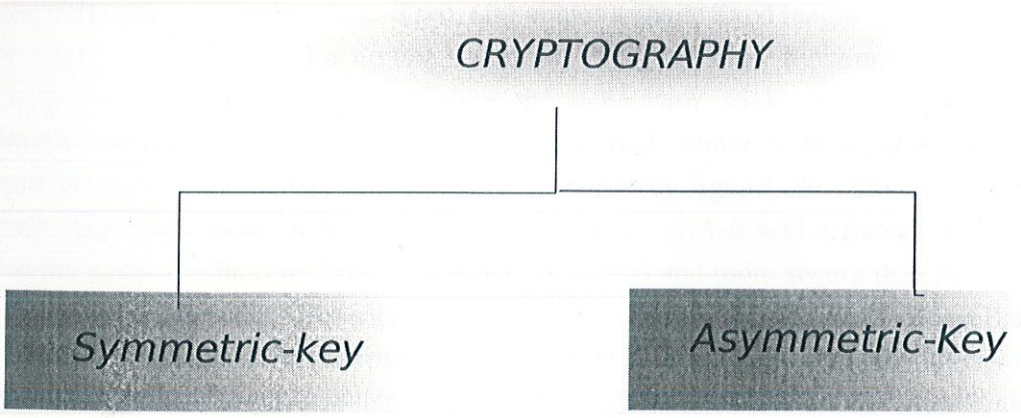


Figure 1: Types of cryptography

1.3.1 Symmetric-key cryptography:

Symmetric-key cryptography refers to encryption methods in which both the sender and receiver share the same key (or, less commonly, in which their keys are different, but related in an easily computable way). This was the only kind of encryption publicly known until June 1976.

Symmetric key ciphers are implemented as either block ciphers or stream ciphers. A block cipher enciphers input in blocks of plaintext as opposed to individual characters, the input form used by a stream cipher.

The Data Encryption Standard (DES) and the Advanced Encryption Standard (AES) are block cipher designs which have been designated cryptography standards by the US government .Despite its deprecation as an official standard, DES remains quite popular; it is used across a wide range of applications, from ATM encryption to e-mail privacy and secure remote access. Many other block ciphers have been designed and released, with considerable variation in quality. Many have been thoroughly broken.

Stream ciphers, in contrast to the 'block' type, create an arbitrarily long stream of key material, which is combined with the plaintext bit-by-bit or character-by-character, somewhat like the one-time pad. In a stream cipher, the output stream is created based on a hidden internal state which changes as the cipher operates. That internal state is initially set up using the secret key material. RC4 is a widely used stream cipher.

Cryptographic hash functions are a third type of cryptographic algorithm. They take a message of any length as input, and output a short, fixed length hash which can be used in (for example) a digital signature. For good hash functions, an attacker cannot find two messages that produce the same hash. MD4 is a long-used hash function which is now broken; MD5, a strengthened variant of MD4, is also widely used but broken in practice. The U.S. National Security Agency developed the Secure Hash Algorithm series of MD5-like hash functions: SHA-0 was a flawed algorithm that the agency withdrew; SHA-1 is widely deployed and more secure than MD5, but cryptanalysts have identified attacks against it; the SHA-2 family improves on SHA-1, but it isn't yet widely deployed, and the U.S. standards authority thought it "prudent" from a security perspective to develop a new standard to "significantly improve the robustness of NIST's overall hash algorithm toolkit." Thus, a hash function design competition is underway and meant to select a new U.S. national standard, to be called SHA-3, by 2012.

Message authentication codes (MACs) are much like cryptographic hash functions, except that a secret key can be used to authenticate the hash value upon receipt.

1.3.2 Asymmetric -key cryptography:

Symmetric-key cryptosystems use the same key for encryption and decryption of a message, though a message or group of messages may have a different key than others. A significant disadvantage of symmetric ciphers is the key management necessary to use them securely. Each distinct pair of communicating parties must, ideally, share a different key, and perhaps each ciphertext exchanged as well. The number of keys required increases as the square of the number of network members, which very quickly requires complex key management schemes to keep them all straight and secret.

In a 1976 paper, Whitfield Diffie and Martin Hellman proposed the notion of *public-key* (also, more generally, called *asymmetric key*) cryptography in which two different but mathematically related keys are used—a *public* key and a *private* key.A public key system is so constructed that calculation of one key (the 'private key') is computationally infeasible from the other (the 'public key'), even though they are necessarily related. Instead, both keys are generated secretly, as an interrelated pair.

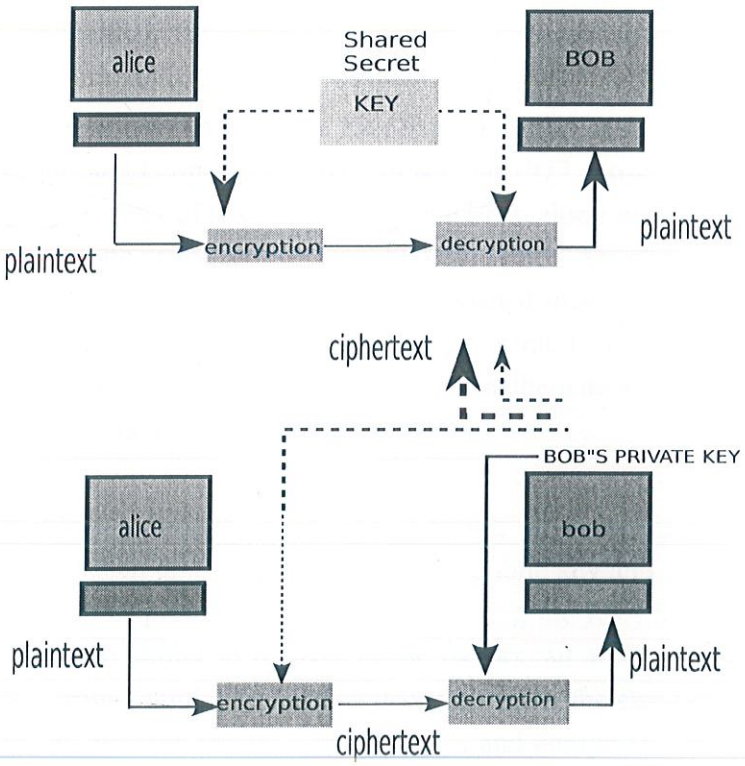


Figure 2: Data transfer using symmetric and asymmetric cryptography

In public-key cryptosystems, the public key may be freely distributed, while its paired private key must remain secret.

In a public-key encryption system, the public key is used for encryption, while the private or secret key is used for decryption.

Diffie and Hellman gave Diffie–Hellman key exchange protocol, a solution that is now widely used in secure communications to allow two parties to secretly agree on a shared encryption keyThis race was finally won in 1978 by Ronald Rivest, Adi Shamir, and Len Adleman, whose solution has since become known as the RSA algorithm.

Public-key cryptography can also be used for implementing digital signature schemes. A digital signature is reminiscent of an ordinary signature; they both have the characteristic of being easy for a user to produce, but difficult for anyone else to forge. Digital signatures can also be permanently tied to the content of the message being signed; they cannot then be 'moved' from one document to another, for any attempt will be detectable. In digital signature schemes, there are two algorithms: one for *signing*, in which a secret key is used to process the message (or a hash of the message, or both), and one for *verification,* in which the matching public key is used with the message to check the validity of the signature. RSA and DSA are two of the most popular digital signature schemes.

1.4 RSA:

RSA is the most widely used Public Key Cryptography .It was introduced by Ron **R**ivest ,Adi **S**hamir and Leonard **A**delman of MIT in 1977.A user of RSA creates and then publishes the product of two large prime numbers, along with an auxiliary value, as their public key. The prime factors must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, if the public key is large enough, only someone with knowledge of the prime factors can feasibly decode the message. Its security is based on following two mathematical problems:

a) factoring large numbers

b) RSA problem

The RSA problem is defined as the task of taking n^{th} roots modulo a composite: recovering a value such that , where is an RSA public key and is an RSA cipher-text. Currently the most promising approach to solving the RSA problem is to factor the modulus. With the ability to recover prime factors, an attacker can compute the secret exponent from a public key , then decrypt using the standard procedure. To accomplish this, an attacker factors into and , and computes which allows the determination of from. No polynomial-time method for factoring large integers on a classical computer has yet been found, but it has not been proven that none exists.

As of 2010, the largest (known) number factored by a general-purpose factoring algorithm was 768 bits long (see RSA-768), using a state-of-the-art distributed implementation. RSA keys are typically 1024–2048 bits long. Some experts believe that 1024-bit keys may become breakable in the near future (though this is disputed);

few see any way that 4096-bit keys could be broken in the foreseeable future. Therefore, it is generally presumed that RSA is secure if is sufficiently large.

1.5 RSA Algorithm:

The RSA algorithm involves three steps: key generation, encryption and decryption-

1.5.1 Key Generation:

1. Choose two distinct prime numbers p and q.
2. Compute n = p*q.

   'n' is used as the modulus for both private and public key.
3. Compute $\varphi(n) = (p-1)(q-1)$

   where φ is Euler's totient function
4. Choose an integer e such that $1 < e < \varphi(n)$ and $\gcd(e,\varphi(n)) = 1$, i.e. e and $\varphi(n)$ are coprime.

   e is released as the public key exponent.
5. Determine $d = e^{-1} \bmod \varphi(n)$; i.e. d is the multiplicative inverse of e mod $\varphi(n)$.

   d is kept as the private key exponent.

1.5.2 Encryption:

Cipher text 'c' is computed using following relation:

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

where m is message in plain text
e is public key of receiver
n is as computed above

1.5.3 Decryption:

Cipher text 'c' is converted into the plain text message 'm' at the receiver side with the private key 'd' of receiver using the following relation :

$$m = c^d \pmod{n}.$$

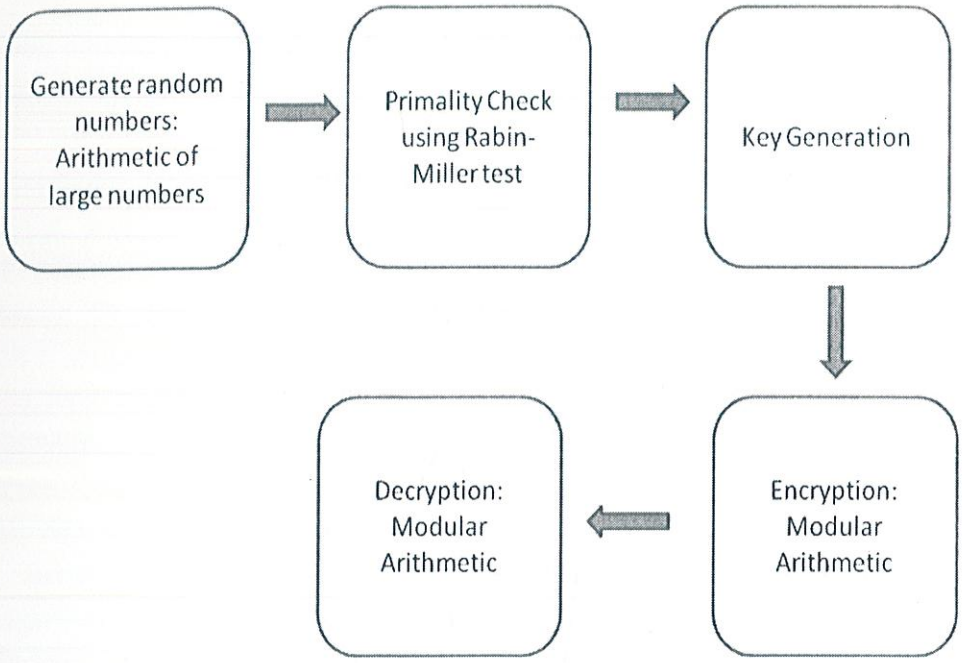


Figure 3: Flowchart of steps involved in RSA implementation

CHAPTER 2: ARITHMETIC OF LARGE NUMBERS

As the security of RSA depends on Large numbers, so our main step is dealing with large numbers.

Every programming language provides the basic arithmetic operation as primitives, these languages include the integer data type supporting the four basic arithmetic operations: addition, subtraction, multiplication and division. These operations are directly mapped to hardware level arithmetic instructions, so size range of integers depends on the underlying processor. We have worked on the PC's a 32 bit machine and use C programming language, which support integer size of 16 bits, even if we use long long int they are of 64 bit size. In RSA, keys are typically 1024–2048 bits long therefore, we develop programs that can handle large bits.

Representing truly enormous integers require stringing digits together. Two possible representations are there :

a) **Array of digits:** In array of digits the initial element of array represents the least significant digit . maintaining the counter with the length of number in digits can aid efficiency by minimizing operations that don't effect the outcome.

b) **Linked Lists of digits:** Using linked list we can dynamically allocate the memory.

We have used array of digits representation for implementing arithmetic operations. Dynamic memory allocation and linked list provide an illusion of being able to get unlimited amount of memory on demand . However linked structures can be really a waste of memory, since part of each node consist of link to another node.

We defined a data type 'large' of 500 decimal digits (1659 bits) :

```
#define MAXDIGITS
#define PLUS                    1       /* positive sign bit */
#define MINUS                   -1      /* negative sign bit */

typedef struct {
char digits[MAXDIGITS];         /* represent the number */
int lastdigit;                  /* index of high-order digit */
} large;
```

Each digit is represented using a single byte character ,this required a lot of care in converting the character into numbers and vice-versa but the requirement of sapce saving was met. The function used to convert the characters of the array into integer :

```
void char_to_large(char *s,large *n)
{
int len=strlen(s);
int l;
for (l=0;l<len;l++)
    n->digits[l]=s[len-l-1]-'0';
printf("Char->Large : Done \n");
n->lastdigit=strlen(s)-1;
}
```

If we use a number having digits less then 500, highest index of array used, it took garbage value. So we had to initialize the array by first filling it with zeros,the function used for this :

```
void initialize_large(large *n)
{
int i;
for (i=0; i<MAXDIGITS; i++)
n->digits[i] =(char)0;
n->lastdigit = -1;
}
```

To eliminate the leading zeros of a number (or result calculated) we used zero_justify function:

```
void zero_justify(large *n)
{
while ((n->lastdigit > 0) && (n->digits[ n->lastdigit ] == 0))
n->lastdigit --;
}
```

The results were printed using following print function :

```
print_large(large *n)
{
int i;
for (i=n->lastdigit; i>=0; i--)
   printf("%c",('0'+ n->digits[i]));
printf("\n");
}
```

2.1 Addition:

It is done by taking the digits having same index ,of the numbers two be added from right to left ,with any overflow rippling to the next field as carry. Flow chart of the function used is :

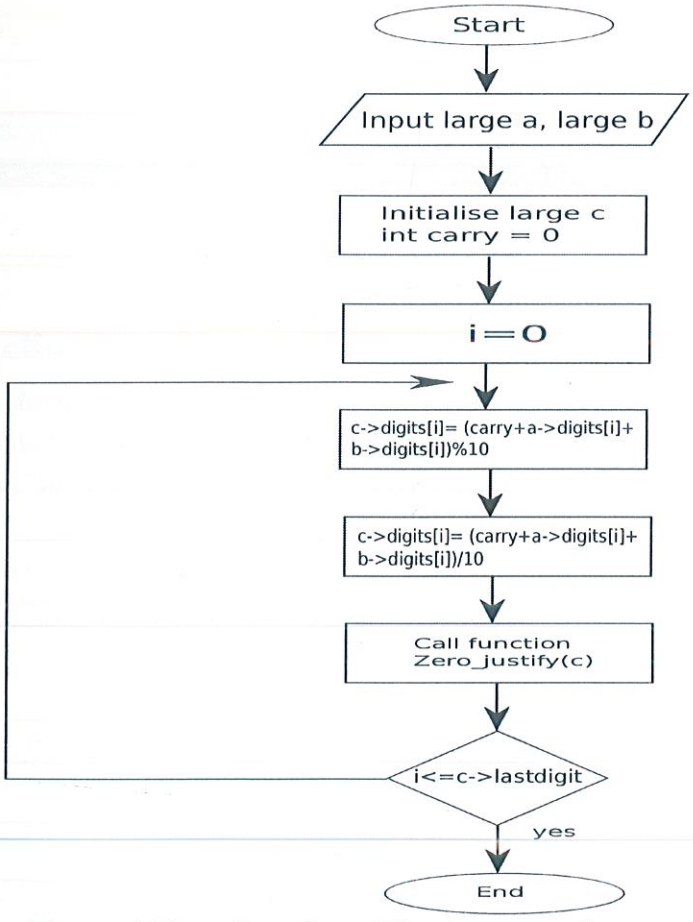


Figure 4:Flow chart for adding large number

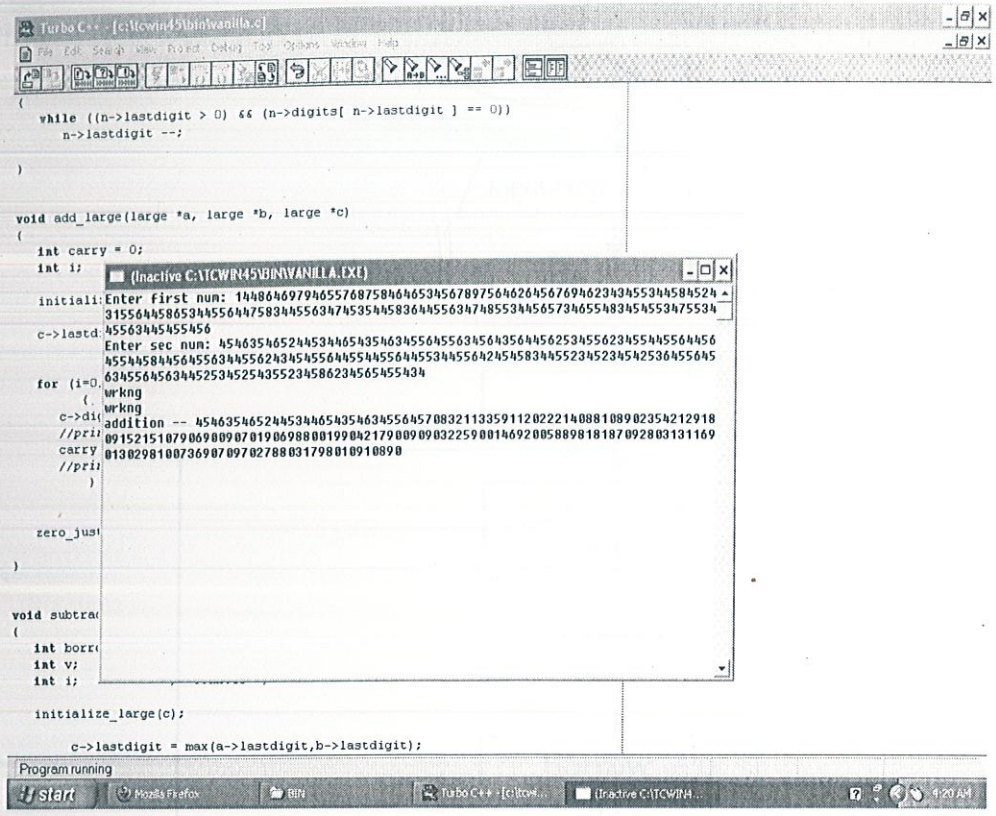


Figure 5: Result of Addition

2.2 Subtraction:

In subtraction ,the main thing of being cared is borrow. To ensure that borrowing terminates, it should be made sure that smaller magnitude number is being subtracted from the larger one. To compare two 'large' data type numbers the function used is :

```
compare_large(large *a, large *b)
{
int i;              /* counter */

if (b->lastdigit > a->lastdigit) return (PLUS);
if (a->lastdigit > b->lastdigit) return (MINUS);

for (i = a->lastdigit; i>=0; i--)
{
if (a->digits[i] > b->digits[i]) return(MINUS);
if (b->digits[i] > a->digits[i]) return(PLUS);
```

```
    }
return(0);}
```

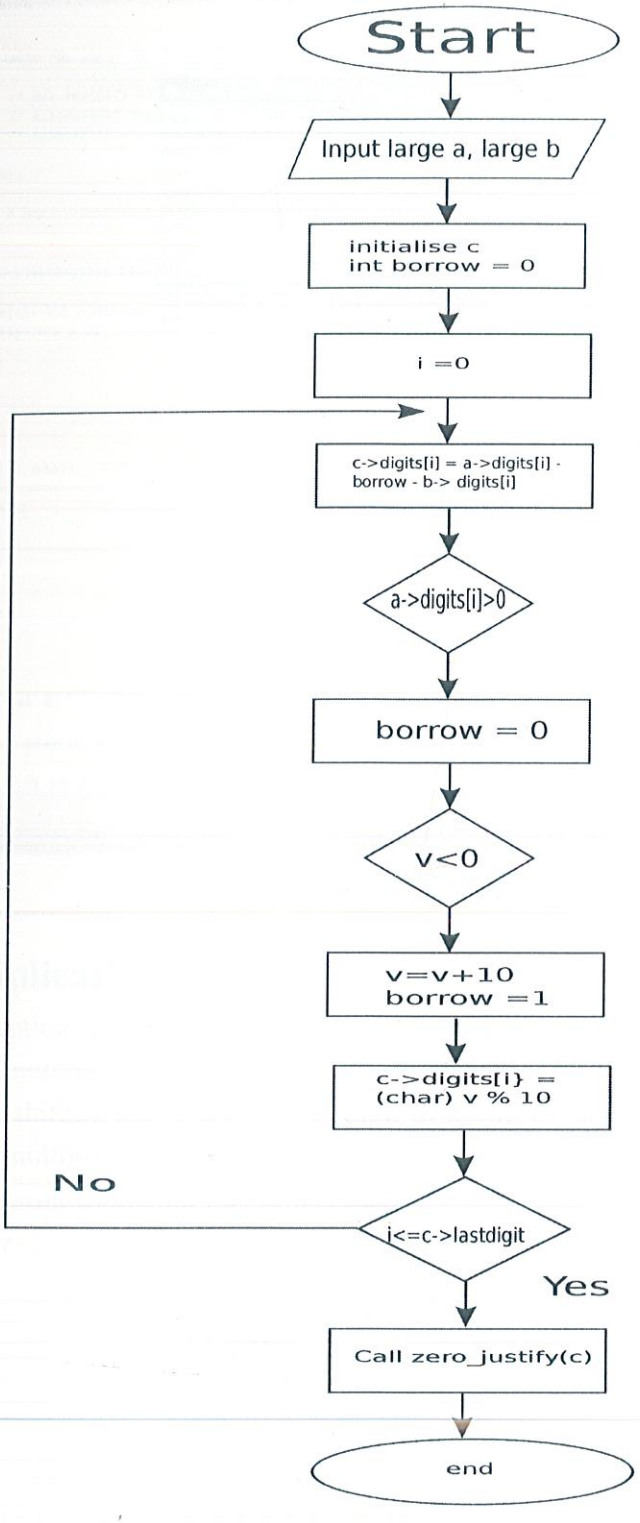


Figure :6:Flow chart for subtracting large number

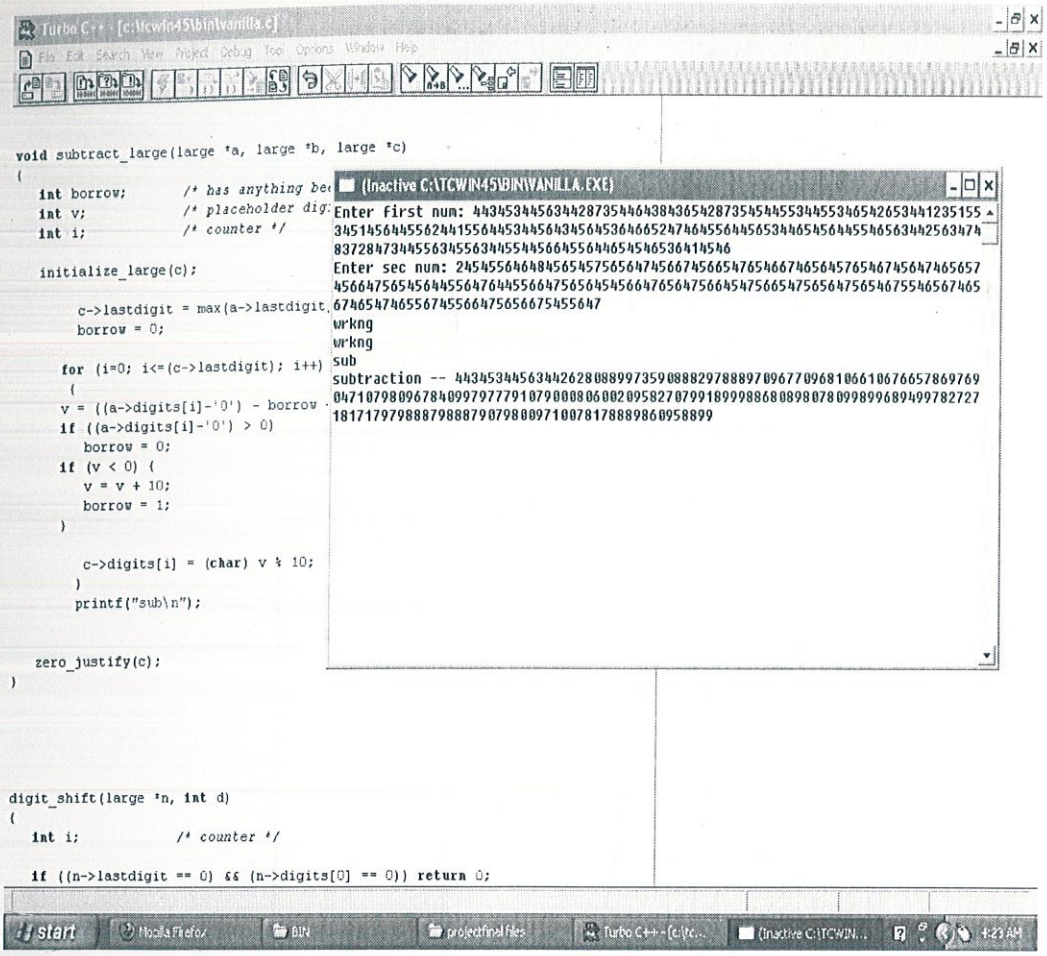


Figure 7: result of subtraction

2.3 Multiplication:

For multiplication row by row method is used, i.e. multiplying first number by the digit of second number. starting from the least significant digit to most significant digit . Keep on shifting the result by one digit to the right and adding this to the result found by multiplying the next digit of second number.

To shift the result following function is used :

```
digit_shift(large *n, int d)
{
int i;                  /* counter */
 if ((n->lastdigit == 0) && (n->digits[0] == '0')) return 0;
for (i=n->lastdigit; i>=0; i--)
n->digits[i+d] = n->digits[i];
for (i=0; i<d; i++) n->digits[i] = 0;
n->lastdigit = n->lastdigit + d;
return 0  ;}
```

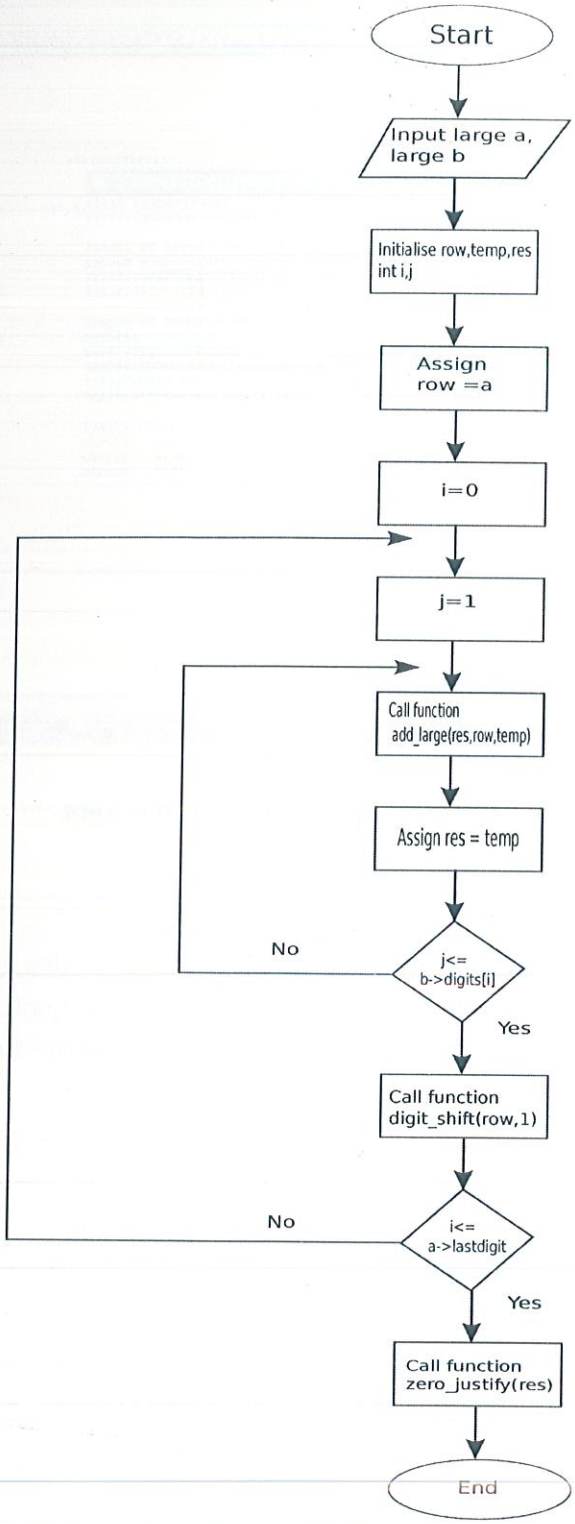


Figure :8:Flow chart for multiplying large number

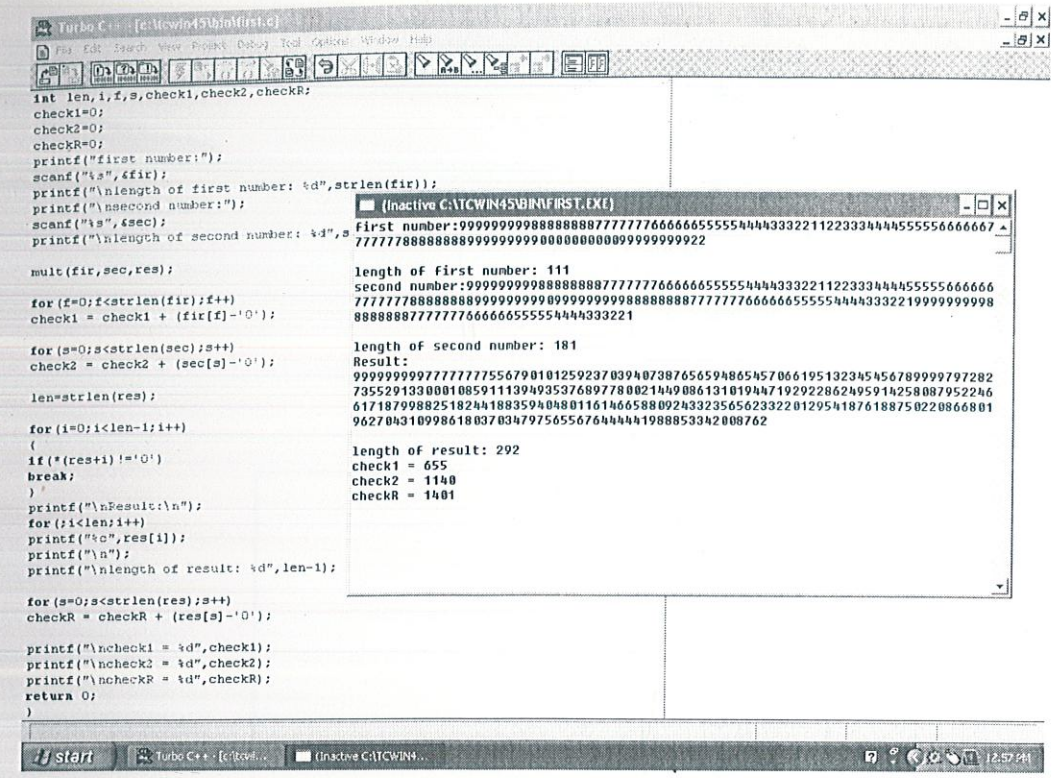


Figure 9: Result of Multiplication

2.4 Division:

Division by repeated subtraction is again far too slow to work with large numbers, but the basic repeated loop of shifting the remainder to the left, including the next digit, and subtracting off instances of the divisor. Pseudo code for division-

```
for (i=a->lastdigit; i>=0; i--) {
    digit_shift(&row,1);
    row.digits[0] = a->digits[i];
    c->digits[i] = 0;
    while (compare_large(&row,b) != PLUS) {
          c->digits[i] ++;
          subtract_large(&row,b,&tmp);
          row = tmp;
    }
}
 *rem = row;
zero_justify(c);
```

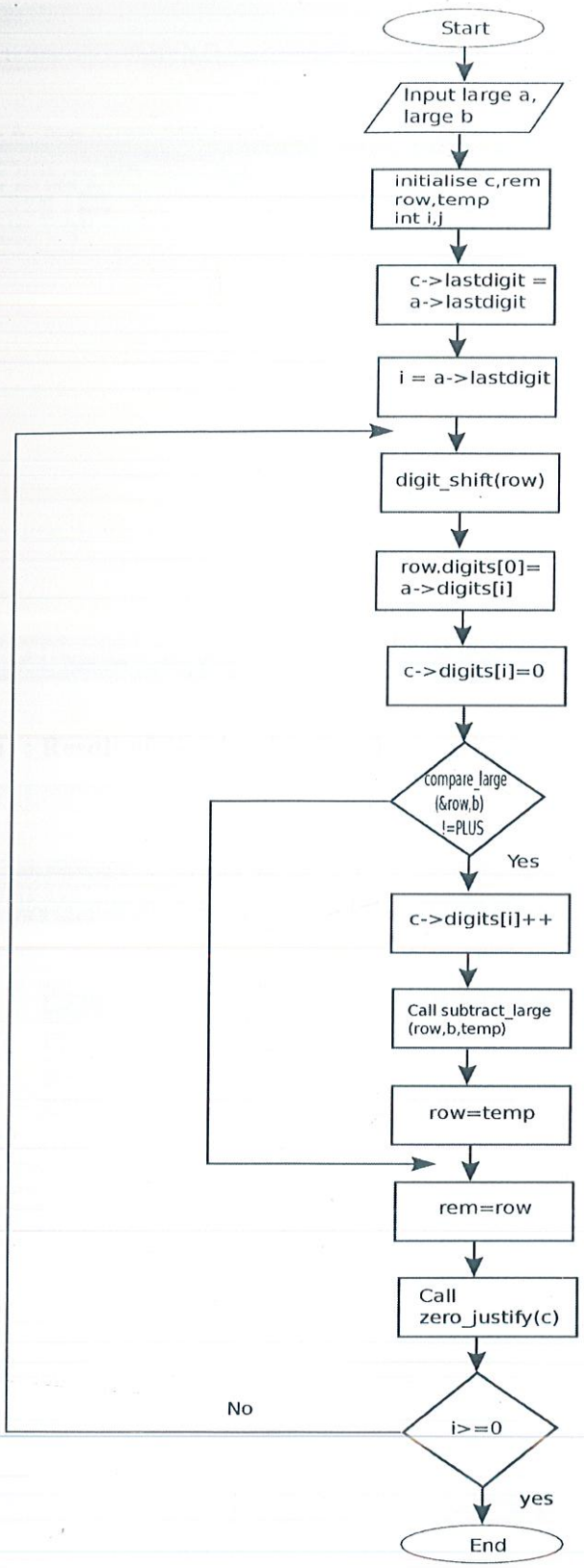


Figure 10: Flow chart for dividing large number

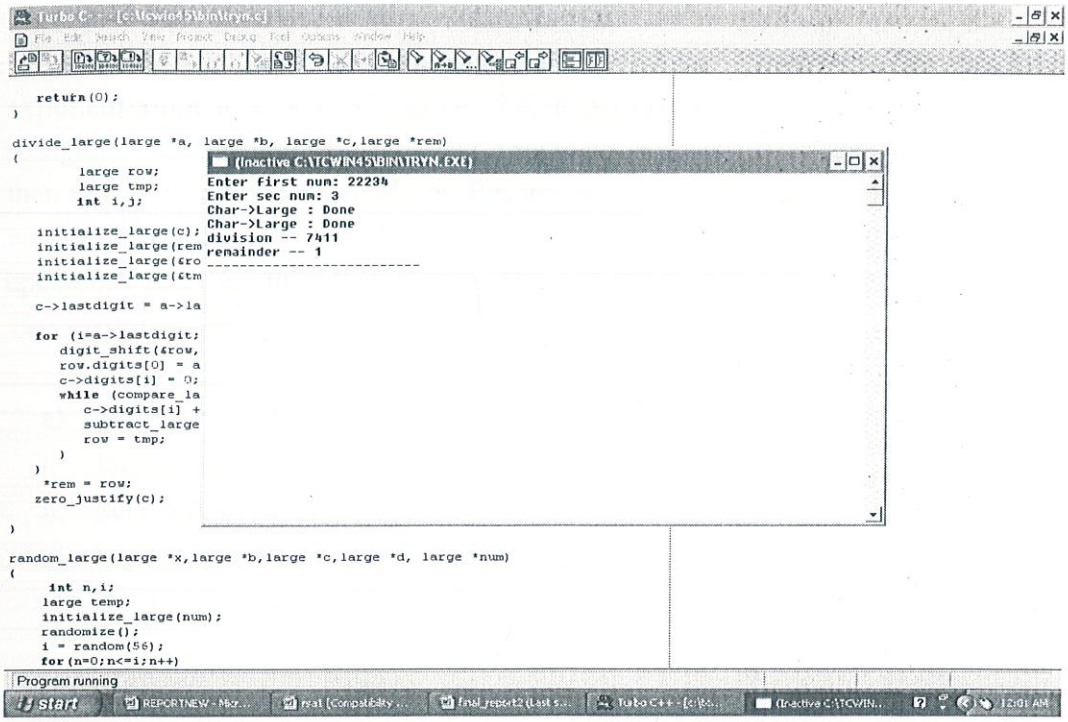


Figure 11 (a) : Result of division for division(small number as input)

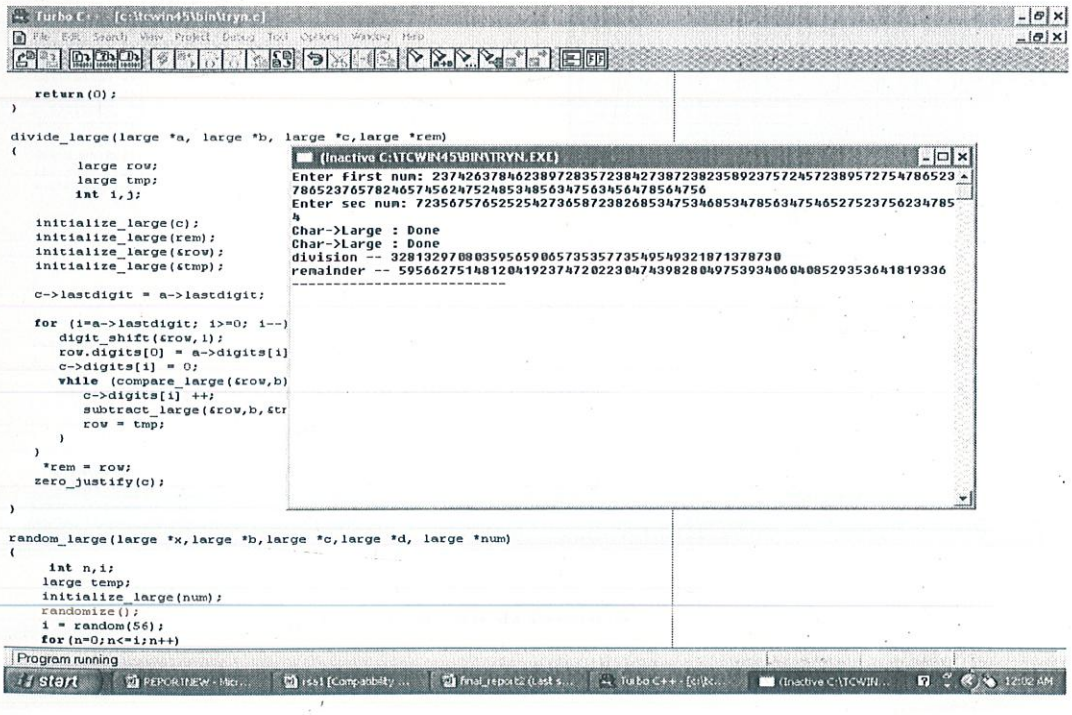


Figure 11(b): Result of division when large numbers were the input

2.5 Exponentiation:

In RSA we have to calculate only modular exponentiation, **Modular exponentiation** is a type of exponentiation performed over a modulus. The most straightforward method of calculating a modular exponent is to calculate b^e directly, then to take this number modulo m. But this method consumes a lot of memory . So we use an algorithm in which the required memory is substantially less, however, operations take less time than before. The end result is that the algorithm is faster. This algorithm used is :

a) Set $c = 1$, $e' = 0$.

i. Increase e' by 1.

ii. Set $c \equiv (b.c) \pmod n$.

iii. If $e' < e$, goto step i. Else, c contains the correct solution to $c \equiv b^e \pmod n$.

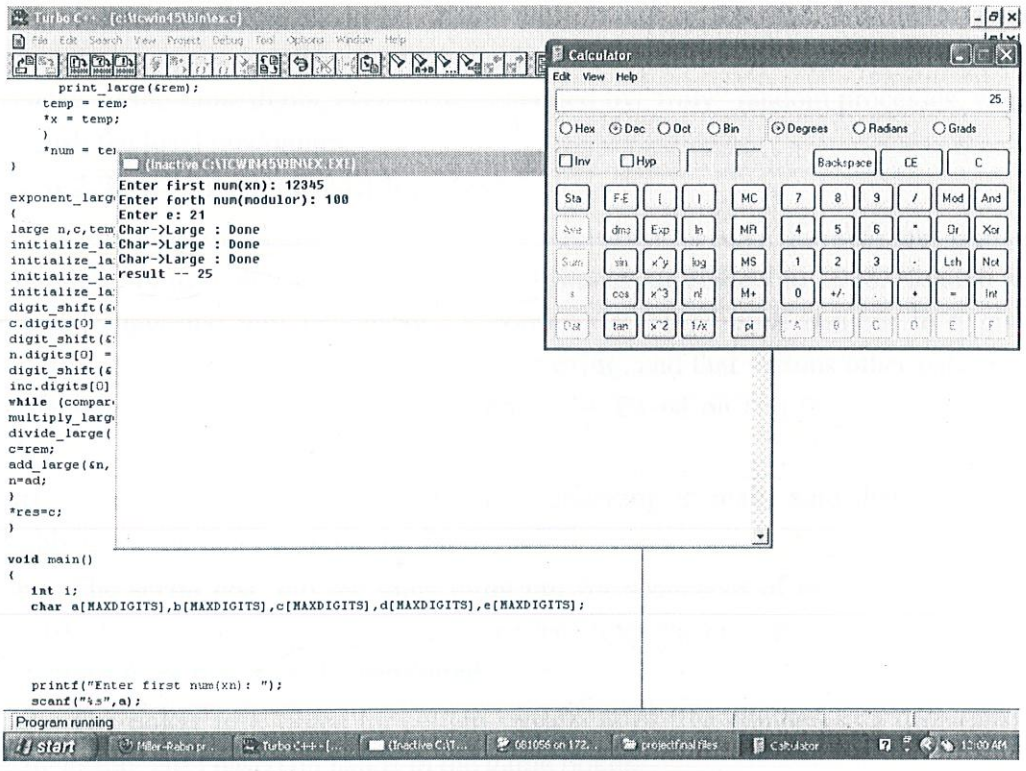


Figure 12: Result of modular exponentiation

2.6 Random Number Generation

To generate keys in RSA implementation with two prime numbers are needed. To obtain large prime numbers first large random number is generated and then primality test is performed.

2.6.1 Randomness of a number:

A numeric sequence is said to be statistically random when it contains no recognizable patterns or regularities. Statistical randomness does not necessarily imply "true" randomness, i.e. objective unpredictability. Pseudo randomness is sufficient for many uses, such as statistics, hence the name statistical randomness. Randomness is of two types:

a) **Global randomness:** It is based on the idea that "in the long run" a sequence looks truly random, even if certain sub-sequences would not look random. e.g.: In a "truly" random sequence of numbers of sufficient length, it is probable that here would be long sequences of nothing but zeros, though on the whole the sequence might be random

b) **Local randomness:** Local randomness refers to the idea that there can be minimum sequence lengths in which random distributions are approximated. Long stretches of the same digits, even those generated by "truly" random processes, would diminish the local randomness.

2.6.2 Tests to check randomness:

The first tests for random numbers were published by M.G. Kendall and Bernard Babington Smith in the Journal of the Royal Statistical Society in 1938. Kendall and Smith's suggested null hypothesis according to which each number in a given random sequence had an equal chance of occurring, and that various other patterns in the data should be also distributed equiprobably. Based on this they suggested four tests :

a) The **frequency test** was very basic: checking to make sure that there were roughly the same number of 0s, 1s, 2s, 3s, etc.

b) The **serial test**, did the same thing but for sequences of two digits at a time (00, 01, 02, etc.), comparing their observed frequencies with their hypothetical predictions were they equally distributed.

c) The **poker test**, tested for certain sequences of five numbers at a time (aaaaa, aaaab, aaabb, etc.) based on hands in the game poker.

d) The **gap test** looked at the distances between zeroes (00 would be a distance of 0, 030 would be a distance of 1, 02250 would be a distance of 3, etc.). If a given sequence was able to pass all of these tests within a given degree of significance (generally 5%), then it was judged to be, in their words locally random.

e) **Autocorrelation test** tests the correlation between numbers and compares the

sample correlation to the expected correlation of zero.

We have used Linear Congruential Method it uses the recurrence

$X_{(n+1)} = (AX_n + B) \bmod m$

where a is a multiplier

b is a increment

m is a modulus

and Xn is a seed

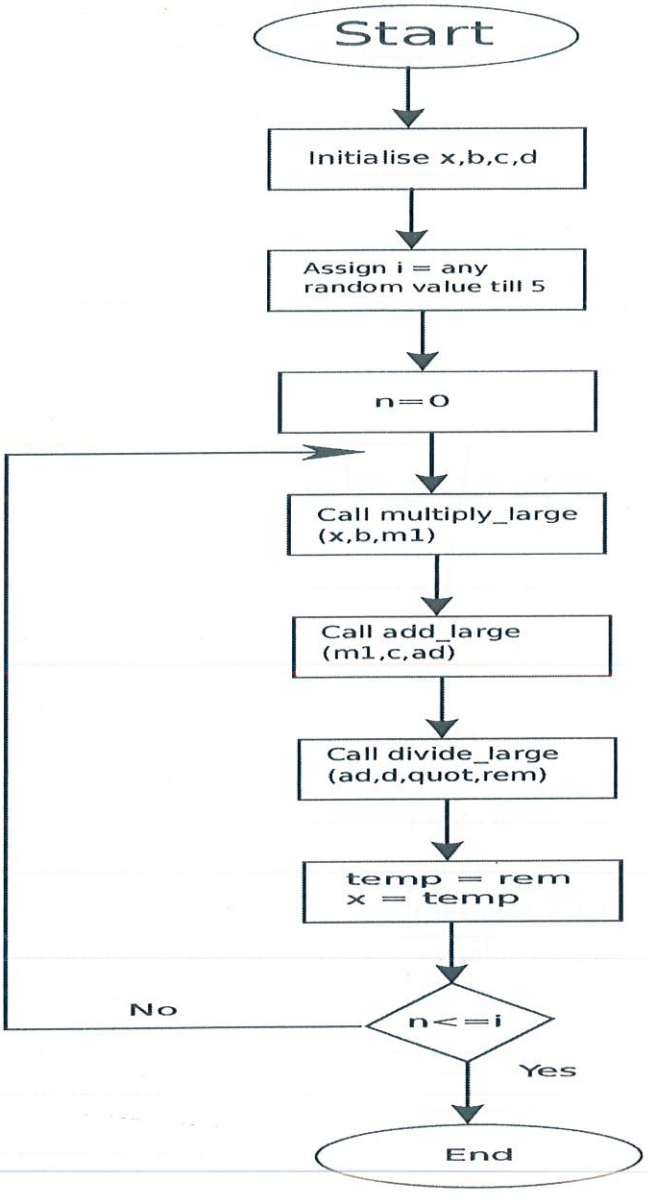


Figure 13: Flow chart for generating random number

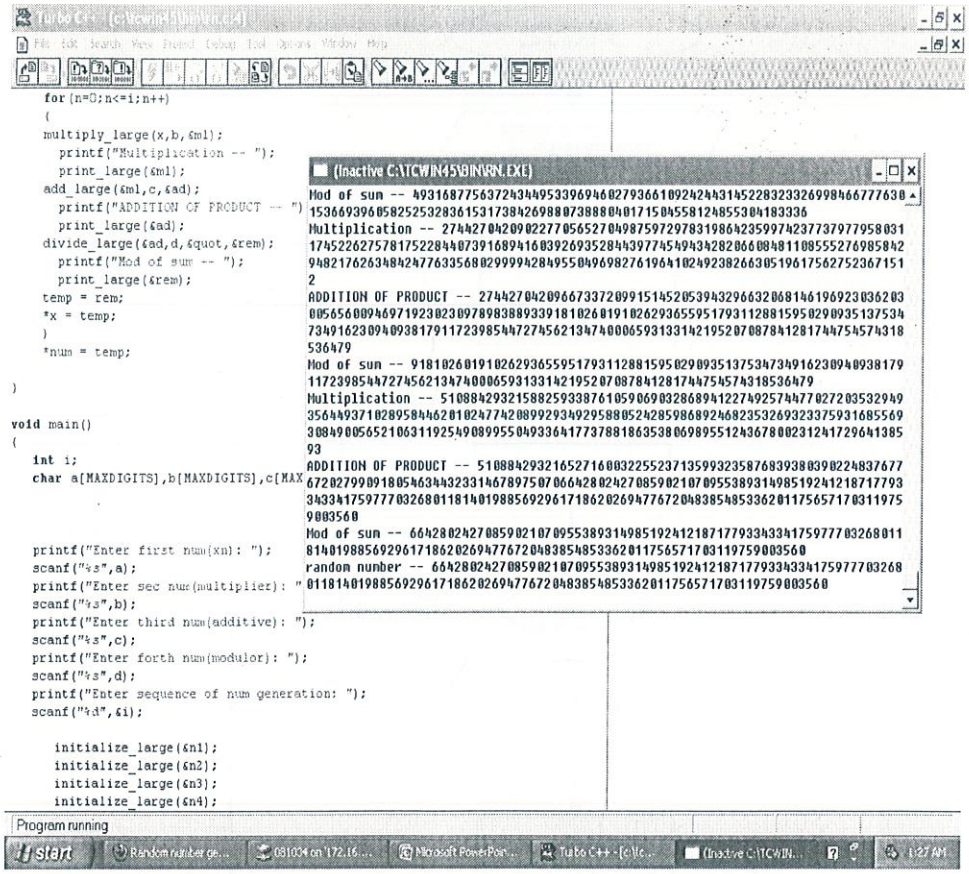


Figure 14: Result of random number generation

2.7 Primality Tests:

Primality test can be divided into two main classes:-

2.7.1 Probabilistic Tests :

These algorithms merely "test" whether *n* is prime in the sense that they declare *n* to be (definitely) composite or "probably prime" .it means that *n* may or may not be a prime number .The Composite numbers which do pass a given primality test are referred to as pseudoprimes. Some probabilistic primality tests are :

a) Miller–Rabin primality test.

b) Solovay–Strassen primality test.

c) Fermat primality test.

2.7.2 Deterministic Tests:

Deterministic algorithms do not erroneously report composite numbers as prime. Some of the examples of deterministic primality tests are elliptic curve primality proving and AKS primality test. Deterministic methods are typically slower than probabilistic ones. In RSA, large prime numbers are required in very few counts of time so probabilistic tests are used. Rabin-Miller test is used here as follows:

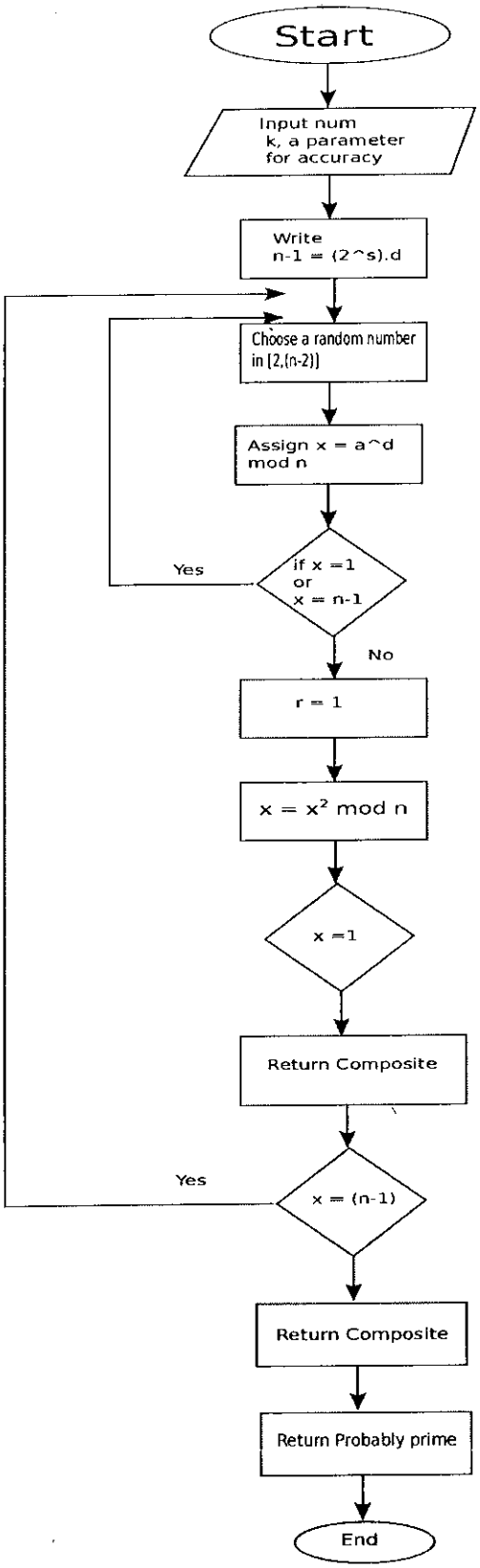


Figure :15:Flow chart for primality test

CHAPTER 3: ENCRYPTION & DECRYPTION

3.1 Encryption:

Plain text message 'm' which is to sent, is converted into cipher text 'c' by the following relation:

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

The method implemented to determine the cipher text uses modular arithmetic. It uses the fact that given two integers a and b, the following two equations are equivalent:

$$c \equiv (a.b) \pmod{m}$$

$$c \equiv (a(b \pmod{m})) \pmod{m}$$

The algorithm is as follows:

1. Set $c = 1$, $e' = 0$.
2. Increase e' by 1.
3. Set $c \equiv (b.c) \pmod{n}$.
4. If $e' < e$, goto step 2. Else, c contains the correct solution to

$$c \equiv b^c \pmod{n}.$$

Example: For b = 4, e = 5, and m = 497, the process is illustrated as:

c′ = 1. c = (1 * 4) mod 497 = 4 mod 497 = 4.

e′ = 2. c = (4 * 4) mod 497 = 16 mod 497 = 16.

e′ = 3. c = (16 * 4) mod 497 = 64 mod 497 = 64.

e′ = 4. c = (64 * 4) mod 497 = 256 mod 497 = 256.

e′ = 5. c = (256 * 4) mod 497 = 1024 mod 497 = 30.

Hence the final solution is 30.

pseudo code of Encryption for e=3 is-

```
for(n=1;n<=3;n++)
{
        multiply_large(b,&temp,&ml);
        divide_large(&ml,m,&quot,&rem);
        temp=rem;
}
*res=temp;
```

The advantage of this method is that, it is memory efficient and operations take less time.

3.2 Decryption

Cipher text 'c' is converted into the plain text message 'm' at the receiver side with the private key 'd' of receiver using the following relation :

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

To calculate the private key exponent 'd' Extended Euclidean algorithm is implemented. D is given by :

$$d = e^{-1} \bmod \varphi(n);$$

i.e. d is the multiplicative inverse of e mod $\varphi(n)$.

d is kept as the private key exponent.

The extended Euclidean algorithm is an extension to the Euclidean algorithm. Besides finding the greatest common divisor of integers a and b, as the Euclidean algorithm does, it also finds integers x and y (one of which is typically negative) that satisfy Bézout's identity

$$ax+by = gcd(a,b)$$

The extended Euclidean algorithm is particularly useful when a and b are coprime, since x is the multiplicative inverse of a modulo b, and y is the multiplicative inverse of b modulo a.

For example, if the gcd of two numbers, 65 and 40 is to be calculated Euclidean algorithm is implemented in the following manner.

Step 1: 65 = 1.40 + 25

Step 2: 40 = 1.25 + 15

Step 3: 25 = 1.15 + 10

Step 4: 15 = 1.10 + 5

Step 5: 10 = 2.5

Therefore, gcd(65,40) = 5

Using method of back substitution,

5 = 15 – 10

= 15 – (25 – 15) = 2.15 – 25

= 2(40 – 25) – 25 = 2.40 - 3.25

= 2.40 – 3(65 – 40) = 5(40) – (3)65

Thus the integers 5 , -3 are respective inverses of the numbers 40 and 65. To calculate 'd' the above method is applied.

Algorithm used to implement the above method of back substitution:

a) Initialize variables r1=phi(n) , r2 = e, s1 = 1, s2 = 0, t1 = 1, t2 = 0

b) While(r2 > 0)

   i. Calculate q as r1/r2

   ii. Calculate r = r1 – q.r2

   iii. Update r1 = r2 , r2 = r

   iv. Calculate s = s1 – q.s2

   v. Update s1 = s2 , s2 = s

   vi. Calculate t = t1 – q.t2

   vii. Update t1 = t2 , t2 = t

   viii. End loop

c) End

Pseudo Code for generating decryption key-

```
while(r2.digits[0]>0 || r2.digits[1]>0)
    {
          divide_large(&r1,&r2,&quotient,&rem);
          multiply_large(&r2,&quotient,&res_r);
          subtract_large(&r1,&res_r,&r);
          r1 = r2;
          r2 = r;
          multiply_large(&l_t2,&quotient,&res_t);
          subtract_large(&l_t1,&res_t,&t);
          l_t1 = l_t2;
          l_t2 = t;
          count++;
}
```

Finally the private key exponent is given as s1 in the algorithm.

3.2.1 Key Generation:

The Public Key is what its name suggests - Public. It is made available to everyone via a publicly accessible repository or directory. On the other hand, the Private Key must remain confidential to its respective owner.

Because the key pair is mathematically related, whatever is encrypted with a Public Key may only be decrypted by its corresponding Private Key and vice versa.

Public key is released as : (e,n)

Where e is the public key exponent and n is the product of two large prime numbers.

Similarly private key is kept with the owner in the form: (d,n)

Where d is the public key exponent and n is the product of two large prime numbers.

Thus if a party A wants to send data to a party B, then the data is encrypted using the public key of B. When the data reaches party B it is decrypted using its private key.

Conclusion

Cryptography is the science used to secure communication. The concept of securing messages through cryptography has a long history. Indeed, Julius Caesar is credited with creating one of the earliest cryptographic systems to send military messages to his generals.

The widespread use of cryptography is limited by key management. In cryptographic systems, the term key refers to a numerical value used by an algorithm to alter information, making that information secure and visible only to individuals who have the corresponding key to recover the information.

A major advance in cryptography occurred with the invention of public-key cryptography. The primary feature of public-key cryptography is that it removes themed to use the same key for encryption and decryption. With public-key cryptography, keys come in pairs of matched "public" and "private" keys. The public portion of the key pair can be distributed in a public manner without compromising the private portion, which must be kept secret by its owner.

The thesis included the generation of these keys using the algorithm RSA. The steps in the algorithm were performed using the library generated which dealt with truly enormous integers.

Further more, sending the message over an electronic network poses several security problems: since anyone could intercept and read the file, you need confidentiality, since someone else could create a similar counterfeit file, the receiver needs to authenticate that it was actually sender who created the file since he could deny creating the file, the receiver needs non-repudiation since someone could alter the file, both sender and receiver need data integrity.

This problem can be solved by using Digital Signatures. The process of digitally signing starts by taking a mathematical summary (called a hash code) of the message. The next step in creating a digital signature is to sign the hash code with your private key. This signed hash code is then appended to the message.

The recipient of message can verify the hash code sent. At the same time, a new hash code can be created from the received message and compared with the original signed hash code. If the hash codes match, then the recipient has verified that the message has not been altered. The recipient also knows that only sender could have sent the message because only he has the private key that signed the original hash code.

Also, one of the major roles of public-key encryption has been to address the problem of key distribution. Further work in this area can be carried forward in two different aspects – distribution of public keys and use of public-key encryption to distribute secret or private keys.

References

1. Network Security and Cryptography by William Stallings.
2. Applied Cryptography: Protocols, Algorithms and Source code in C by Schneier, Bruce.
3. Mathematics of Ciphers: Number Theory and RSA Cryptography by S.C. Coutinho.
4. www.stackoverflow.com

APPENDIX-1

Code for Arithmetic Operations :

```
add_large(large *a, large *b, large *c)
{
int carry = 0;
int i;
initialize_large(c);
c->lastdigit = nmax(a->lastdigit,b->lastdigit)+1;
for (i=0; i<=(c->lastdigit); i++)
{
c->digits[i] =  (char)((carry+(a->digits[i])+(b->digits[i])) % 10);
carry = ((carry + (a->digits[i]) + (b->digits[i]))) / 10;
 }
zero_justify(c);
}


subtract_large(large *a, large *b, large *c)
{
    int borrow ;
    int v;                                  /* placeholder digit*/
    int i;                                        /* counter */
     initialize_large(c);
      c->lastdigit = nmax(a->lastdigit,b->lastdigit);
      borrow = 0;
  for (i=0; i<=(c->lastdigit); i++)
 {
    v = (a->digits[i] - borrow - b->digits[i]);
    if (a->digits[i] > 0)
    borrow = 0;
    if (v < 0) {
    v = v + 10;
    borrow = 1;
```

```
}
 c->digits[i] = (char) v % 10;
 }
zero_justify(c);
}


digit_shift(large *n, int d)
{
int i;                                          /* counter */
if ((n->lastdigit == 0) && (n->digits[0] == '0')) return 0;
for (i=n->lastdigit; i>=0; i--)
 n->digits[i+d] = n->digits[i];
 for (i=0; i<d; i++) n->digits[i] = 0;
 n->lastdigit = n->lastdigit + d;
  return 0  ;
}

void multiply_large(large *a, large *b, large *res)
{
    large row;                          /* represent shifted row */
    large tmp;                          /* placeholder large */
    int i,j;                            /* counters */

    initialize_large(res);
    row = *a;

    for (i=0; i<=a->lastdigit; i++) {
    for (j=1; j<=(b->digits[i]); j++) {
          add_large(res,&row,&tmp);
         *res = tmp;
    }
    digit_shift(&row,1);
    }
zero_justify(res);
}
```

```
compare_large(large *a, large *b)
{
    int i;                                  /* counter */

    if (b->lastdigit > a->lastdigit) return (PLUS);
    if (a->lastdigit > b->lastdigit) return (MINUS);

    for (i = a->lastdigit; i>=0; i--) {
    if (a->digits[i] > b->digits[i]) return(MINUS);
    if (b->digits[i] > a->digits[i]) return(PLUS);
    }

    return(0);
}

divide_large(large *a, large *b, large *c,large *rem)
{
 large row;
 large tmp;
 int i,j;

initialize_large(c);
initialize_large(rem);
initialize_large(&row);
initialize_large(&tmp);

c->lastdigit = a->lastdigit;

for (i=a->lastdigit; i>=0; i--)
 {
      digit_shift(&row,1);
      row.digits[0] = a->digits[i];
      c->digits[i] = 0;
      while (compare_large(&row,b) != PLUS) {
       c->digits[i] ++;
       subtract_large(&row,b,&tmp);
       row = tmp;
```

```
    }
        }
         *rem = row;
        zero_justify(c);
       }


exponent_large(large *b,large *m,large *e,large *res)
{
large n,c,temp,inc;

 initialize_large(&n);
 initialize_large(&c);
 initialize_large(&inc);
 initialize_large(&temp);
 digit_shift(&c,1);
 c.digits[0] =1;
 digit_shift(&n,1);
  n.digits[0] =1;
 digit_shift(&inc,1);
 inc.digits[0] =1;

while (compare_large(&n,e)!= MINUS)
{
    multiply_large(b,&c,&ml);
    divide_large(&ml,m,&quot,&rem);
    c=rem;
    add_large(&n,&inc,&ad);
    n=ad;
   }
*res=c;
}
```

Code for Random number geration :

```
random_large(large *x,large *b,large *c,large *d, large *num)
{
    int n,i;
    large temp;
    initialize_large(num);
    randomize();
    i = random(56);

     for(n=0;n<=i;n++)
       {
       multiply_large(x,b,&ml);
       add_large(&ml,c,&ad);
       divide_large(&ad,d,&quot,&rem);
       temp = rem;
       *x = temp;
       }

      if((temp.digits[0])%2==0)
        {
              temp.digits[0] =temp.digits[0]+1;
        }
            *num = temp;
       printf("random number->Done");
      }
```

Code for Prime Number Testing:

```
test_prime(large *n,int k)
{
    long int m,i,r;
    large temp,od,remain,exp,two,rndno,nm1;
```

```
    initialize_large(&od);
    initialize_large(&two);
    digit_shift(&two,1);
    two.digits[0] =2;
    zero_justify(&two);
    one.digits[0] =1;

    initialize_large(&temp);
    initialize_large(&exp);
    initialize_large(&rndno);
    initialize_large(&nm1);
    temp = *n;
    subtract_large(&temp,&one,&nm1);
    printf("\nn-1=");
    print_large(&nm1);
    m=0;  //assign count 2^s.d,s=m

  divide_large(&nm1,&two,&od,&remain);
  print_large(&od);
  print_large(&remain);
  while(remain.digits == NULL)
  {
     nm1=od;
     divide_large(&nm1,&two,&od,&remain);
     m++;
}
  temp = od;
  for(i=0;i<k;i++)
   {
     print_large(&nm1);
     random_large(&n1,&n2,&n3,&nm1,&rndno);
     printf("\nrndno.=");
     print_large(&rndno);
     printf("\norgno.=");
     print_large(n);
     exponent_large(&rndno,&od,n,&exp);
     printf("\nx=");
```

```
  print_large(&exp);
if (((compare_large(&exp,&one)!= MINUS) && (compare_large(&exp,&one)!
=     PLUS))     ||     ((compare_large(&exp,&nm1)!=     MINUS)     &&
(compare_large(&exp,&nm1)!= PLUS)))
  {
       printf("\nifcoming");
      continue;
       }
      else
   {
        printf("\nelsecoming");
       for(r=1;r<=(m-1);r++)
   {

 exponent_large(&exp,&two,n,&temp);
 exp=temp;
  }

if(((compare_large(&exp,&one)!= MINUS) && (compare_large(&exp,&one)!=
PLUS)))
 {
   printf("\ncomposite");
    return -1;
 }
if(((compare_large(&exp,&nm1)!= MINUS) && (compare_large(&exp,&nm1)!=
PLUS)))
  {
    continue;
  }
}

  printf("\ncompo");
  return -1;
}
  printf("\nprime");
   return 1;
  }
```

Code for Encryption :

```
encryption_large(large *b,large *m,large *res)
{
      int n;
      large one,temp;
      initialize_large(&one);
      initialize_large(&temp);
      digit_shift(&one,1);
      one.digits[0] =1;

      temp = one;
      for(n=1;n<=3;n++)    /* for e =3
      {
      multiply_large(b,&temp,&ml);
      divide_large(&ml,m,&quot,&rem);
      temp=rem;
      }
      *res=temp;
}
```

Code for Decryption :

```
decrypt_large(large *num,large *exp)
{
int flag =-1;

initialize_large(&r1);
initialize_large(&r2);

r1 = *num;
r2 = *exp;

while(r2.digits[0]>0 || r2.digits[1]>0)
```

```
{
 divide_large(&r1,&r2,&quotient,&rem);
  multiply_large(&r2,&quotient,&res_r);
 if(compare_large(&res_r,&r1)==MINUS && flag==1)
 {
 subtract_large(&r1,&res_r,&r);
 flag=1; // +x
 }
 else if(compare_large(&res_r,&r1)==PLUS && flag==1 )
 {
 subtract_large(&r1,&res_t,&r);
 flag=-1; // -x
 }

else
if((compare_large(&res_r,&r1)==PLUS                                    ||
compare_large(&res_r,&r1)==MINUS) && flag==-1 )
{
add_large(&r1,&res_r,&r);
flag=-1;
}
subtract_large(&r1,&res_r,&r);
r1 = r2;
r2 = r;

printf("quotient :" );
print_large(&quotient);
multiply_large(&l_t2,&quotient,&res_t);
if(compare_large(&res_t,&l_t1)==MINUS && flag==1)
  {
   subtract_large(&l_t1,&res_t,&t);
   flag=1; // +x
   }
else
 if(compare_large(&res_t,&l_t1)==PLUS && flag==1 )
{
  subtract_large(&res_t,&l_t1,&t);
```

```
    flag=-1; // -x

    }

else
if((compare_large(&res_t,&l_t1)==PLUS                                    ||
compare_large(&res_t,&l_t1)==MINUS) && flag==-1 )
{
 add_large(&res_t,&l_t1,&t);
 flag=-1;//-x
}

l_t1 = l_t2;
l_t2 = t;

count++;
}
 printf("gcd\n");
 print_large(&r1);
 print_large(&l_t1);
}
```