**Report on RSA Algorithm**

**Abstract.**

This is a report of the analysis of the RSA algorithm implemented in this project. It also presents asymptotic analysis on the running time of the algorithms used in this program.

**Introduction.**

RSA is a public key cryptosystems used to transfer data securely. RSA stands for Ron Rivest, Adi Shamir, and Leonard Adleman who publicly described the algorithm in 1977 at MIT. RSA Algorithm implementation involves generation a public key and private keys using two large randomly generated prime numbers. The Algorithm encrypts the message using public key. No one can decrypt the encrypted message except the user possessing the private key.

**RSA Cryptosystem**

Take two large Prime numbers p and q

Compute their product n: n = p \* q

Select an odd integer e that is relatively prime to

pi( n) = (p-1)(q-1);

Compute d as the multiplicative inverse of e, modulo pi(n);

(de = 1 mod pi(n))

Publish P = (e, n) as the RSA public key;

Keep secret S = (d, n) as the RSA secret key.

The encrypted message P (M) = M^e mod n.  
The decrypted message S (C) = C^d mod n, C =P (M).

**Analysis of Implementation**

The RSA algorithm is implemented using JAVA in four main stages:

1.Random Prime Generation

2.Public and private key generation

3.Encryption

4.Decryption

**1.Random Prime Generation**

To generate a random prime the Java program uses Big Integer class to enable the handling of very large numbers.

*randomPrimeGenerator() :* Using the length of prime number as the input this function returns a random prime number.  
This function calculates the range of numbers in which a random number is to be picked. For example if 3 is the length of the prime then the range is calculated as shown:   
[10pow(3-1), 10pow(3)-1]=[100,999] ~ all three-digit numbers.  
From this range, considering a random number of a particular bit length a random big Integer is selected. The generated random number is to be checked for the primality using the Fermat’s primality test.   
*fermatsTest():* This function takes the generated random number as input and returns a Boolean value true or false indicating the primality of the random number. The random prime generator calls this function until it generates a random Prime.

**2.Public and private key generation**

Public key (e, n):  
The value of n=P\*Q and phiOfN=(p-1)\*(q-1) is calculated by using big Integer multiply function. The value of e is selected such that it is an odd number relatively prime to phiOfN i.e. GCD (phiOfN, e)=1.

Private key (d, n):  
*extendedEclideanAlg():* This function is implementation of extended Euclidean Algorithm to calculate d, a multiplicative inverse of e and 1 mod phiOfN.  
This function takes phiOfN and e as the input and outputs a quotient q and d such that gcd (phiOfN, e)=1= phiOfN \* q+ e \* d.

To calculate GCD (a, b), the extended Euclidean algorithm calculates x and y such that GCD= a\* x+ b\*y. In its each recursive call computes the quotient and remainder of division. For example worst case when GCD is 1:

GCD (a, b)= GCD (b, a mod b).

GCD (38, 11)= GCD (11, 38 mod 11)=GCD (11, 5)

= GCD (5, 11 mod 5)= GCD (5, 1)

=GCD (1, 5 mod 1)=GCD (1, 0)

=1

We can deduce that after two operations the first value reduces by a factor of ½. The no of iterations before a value b is 0 depends on bit size of a i.e. log a and bit size of b i.e. log b. Hence the no of iterations grows logarithmically with size of inputs. The overall computation is achieved in O (log a + log b) time. The below figure shows a scatter plot considering time of execution of extended Euclidean function Vs. Input bit length for a sample execution data.



**3.Encryption**

*encryptRSA():* This function takes input message and encrypts it using the public key (e, n).  
This function takes the ASCII value of each character in the message and converts the message into a single integer representation.  
For example if the message is ‘ab’ then it takes the ASCII value of each character and represents as a single integer.   
Since ASCII values of a and b are, a=97 and b=98   
function computes 97\* 128^1+98\*128^0=12514.The time complexity in calculating the integer is O (n^2) since it involves a multiplication. This message is passed to mod exponent function.  
  
modExponent(): This function calculates (message)^e mod n value using repeated squaring algorithm. The output derived will be the encrypted message.

To calculate x^y mod m, the value of y reduces by half in each iteration so no of iterations will be in time O (log y). Each iteration also calculates modular multiplications. The size of integer multiplied in a modular multiplication will be less than m. So the multiplication of two integers less than m will be done in O (log m \* log m) time. The overall computation is achieved in O (log y \* log m \* log m) time. If this is computed on k different values the time complexity will be O (k\*log y \* log m \* log m).  
In the program we have computed for two different values 2 and 3.So The complexity is  
O (2\*log y \* log m \* log m).

**4.Decryption**

decryptRSA(): This function takes input as the encrypted message and decrypts it using the private key (d, n).  
The function initially stores the input message in a big integer, calculates (message)^e mod n using mod Exponent function. It derives the each character letter of the message and stores it in a character array. The entries in the character array that are stored in the reverse sequence are concatenated to a string variable in the required sequence.

For example if the message after decryption is = 12514.Then character an attribute 12514 mod 128 =98 which is ASCII value of character ‘a’  
12514/128=97 which is ASCII value of character ‘b’. The value is of ach character is stored into a character array. The contents of character array are concatenated to a string. Thus the original message ab is derived.

**References**

[1]. <http://en.wikipedia.org/wiki/RSA_(algorithm>)

[2]. <http://www.cs.berkeley.edu/~vazirani/algorithms/chap1.pdf>

[3]. <http://www.cs.uakron.edu/~zduan/class/435/Notes.htm>