RSA

The RSA algorithm

The RSA algorithm's success is predicated on the computational complexity of inverting a certain function: the function that maps two large primes to their product. It is, indeed, that simple. Factorisation of integers is an NP-complete problem, which does not, *per say*, guarantee its difficulty for large composite integer inputs, but some conditions imposed on the choice of the primes that helps:

a) The prime numbers chosen are not too close together

b) The product is really large- usually about 2^8 bytes long

The reason for b) is, of course, the NP completeness of factorisation. But the reason for a) is much more interesting. John D. Cook illustrates: (full thing at https://www.johndcook.com/blog/2018/10/28/fermat-factoring/)

" The answer goes back to Fermat (1607–1665). His factoring trick is to start with an odd composite n and look for numbers a and b such that

$n = a^2 - b^2$

because if you can do that, then

$n = (a + b)(a - b).$

This trick always works ... but it's only practical when the factors are close together. If they are close together, you can do a brute force search for a and b. But otherwise you're better off doing something else."

I. Computation of keys

1. First, 2 prime numbers p,q of the sender's choice are taken and multiplied together to yield n. Take care to comply with the conditions mentioned above. (As it happens, we are only implementing a "kid's" version of this, so me need not worry too much about this.)

```
//I.1.Read primes
int p=
Integer.parseInt( br.readLine() );
int q=
Integer.parseInt( br.readLine() );
```

```
int n= p*q;
```

The value n will be made public, but p,q are private.
2. Find φ_n= *(p-1)(q-1)*. Then choose, randomly an e which satisfies: $0 < e < \varphi_n$ and gcd(e,φ_n)=1. This can be obtained by seaching 2,...,φ_n for a relatively prime number. These conditions mean that p,q cannot be 2 and 3.

```
//I.2.Get φ and e
int phi_n= (p-1)*(q-1);

//To find 'e':
int e=1 /*Initialise to random value*/,
i /*Index for searching 2,...,phi_n*/,
m, N, r=1 /*Temporary variables*/;

for( i=2; i< phi_n; i++ )
{
if( gcd( i,phi_n )==1 )
{ e= i;}
}
```

3. Compute multiplicative inverse d of e modulo φ_n: $d= e^{-1} \ mod \ \varphi_n$.

```
//I.3.Find e^-1 mod φ
//Search for d such that ed%phi_n is unity
int d=1;
for( i=0; i< phi_n; i++ )
{ if( (i*e)%phi_n ==1 )
```

```
{ d= i;
break; }
}
```

II. Encryption/Decryption

II.1 Encryption

1. Read the "Plaintext" as an integer m.

```
//II.1.1 Read plaintext
int M=
Integer.parseInt( br.readLine() );
```

2. Compute c as m raised to e modulo n: $c = m^e \ mod \ n$. (Keep in mind that if we had taken m^e instead of $m^e \ mod \ n$, it would have also been fine. But then in the practical case, if we had taken a large n, we would we would have obtained, a number of maximum size $\tilde{}\log_2 m^n$, instead of a maximum size of $\tilde{}\log_2 n$.)

```
//II.1.2. Find c=m^e mod n
double c=
Math.pow( M,e ) %n;
```

3. Print as "Encrypted text".

```
//II.1.3. Encrypt
System.out.println(
"Encrypted text is:" + c
```

```
);
```

II.2 Decryption

1. Compute dec as c raised to d modulo n: $dec = c^d \ mod \ n$.

```
//II.2.1.Find dec= c^d mod n
double dec=
Math.pow( c,d ) %n;
```

2. Print as "Decrypted text".

```
C
//II.2.2.Decrypt
System.out.println(
"Decrypted text is:" + dec
);
```

APPENDIX.

A. Computation of GCD

Since Java does not have any provision for goto statements, we may use the very simple recursive version of Euclid's algorithm.

To find gcd(A,B):

1. If A mod B =0, B is the answer.

```
//If end, stop
if( A%B==0 ) return B;
```

2. Else, compute gcd(B, A mod B)

```
//Else, reduce
else return gcd( B, A%B);
```

B. Java Syntax

The above program has omitted the header and other details that go into the real program. In order to compile the program it must be placed in a well-formed Java source file, as below.

```
import java.io.*;

public class RSA
{

public static int gcd( int A, int B )
{
%//If end, stop
%//Else, reduce
}

public static void main( String A[] ) throws
IOException
{

BufferedReader br=
new BufferedReader(
new InputStreamReader( System.in )
);
```

%//I.1.Read primes
%//I.2.Get φ and e
%//I.3.Find e^{-1} mod φ

%//II.1.1 Read plaintext
%//II.1.2. Find $c=m^e$ mod n
%//II.1.3. Encrypt

%//II.2.1.Find dec= c^d mod n
%//II.2.2.Decrypt

}

}