```
/*
Copyright 2015 ALY SHMAHELL

 This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef ENDABI_RSA_CORE_INCLUDED
#define ENDABI_RSA_CORE_INCLUDED
#include <stdio.h>
#include <string.h>
#include <sstream>
using namespace std;
#define WORD 200

/* This struct is used to store the public key sequence and return it as a whole aft
er Public-Key generation
The sequence is : the suitable Public-Key, the corrected Prime p, the corrected Prim
e q, the Modulus n = p*q  */
template <typename typename_> struct pub_key_sequence
{
    typename_ pk, p_, q_, n;
};

/* The phi(modulus) function denotes the number of coprimes the encryption modulus (
modulus = n) has */
template <typename typename_> typename_ phi(typename_ p,typename_ q)
{
    return ((p-1)*(q-1));
}

/* The Greatest Common Divisor function building on the Eucledian Algorithm */
template <typename typename_> typename_ gcd(typename_ r0,typename_ r1)
{
    return (r1==0)?r0:gcd(r1,r0%r1);
}

/* Modular Exponentiation function, using the famous & fast (square and multiply)  m
ethod */
template <typename typename_>
typename_ modular_exponentiation(typename_ base,typename_ exponent,typename_ modulus
_)
{
    typename_ temp;
    if(exponent==1)
        return base;
    else
    {
        if(exponent%2==0)
        {
            temp = modular_exponentiation(base,exponent/2,modulus_);
            return ((temp*temp)%modulus_);
        }
        else
        {
            temp = modular_exponentiation(base,(exponent-1)/2,modulus_);
            temp*=temp;
            temp*=base;
            return (temp%modulus_);
        }
    }
}
```

```
/* This function finds the Modular Inverse using the Extended Eucledian Algorithm.
The modular Inverse is described as  : (a*a^-1 = 1 mod modulus) where (a^-1) is
the inverse of a.  */
template <typename typename_> typename_ EEA(typename_ r0,typename_ r1)
{
    typename_ t;
    typename_ t0=(typename_)0;
    typename_ t1=(typename_)1;
    typename_ temp;
    typename_ r2,r3;
    if(r0<r1)
    {
        temp=r0;
        r0=r1;
        r1=temp;
    }
    r2=r0;
    r3=r1;
    while(r3!=0)
    {
        if(r0>r1)
        {
            t=t0-((r2-(r2-r3*(r2/r3)))/r3)*t1;
            t0=t1;
            t1=t;
        }
        temp = r3;
        r3 = (r2-r3*(r2/r3));
        r2 = temp;
    }
    if(t0<0) t0+=r0;
    return t0;
}

/* a Wrapper for the previous function, this one finds the inverse of the Public-Key
 over Phi(modulus) */
template <typename typename_> typename_ modular_inverse(typename_ a,typename_ p,type
name_ q)
{
    typename_ phi_=phi(p,q);
    return EEA(a,phi_);
}

/*this function encrypts a message character by exponentiating the ascii representat
ion of the character
to the power public-key and reducing the result modulo modulus */
template <typename typename_>
typename_ encrypt(typename_ message,typename_ public_key,typename_ modulus_)
{
    return modular_exponentiation(message,public_key,modulus_);
}

/*this function encrypts a message character by exponentiating the ascii representat
ion
of the character to the power private-key and reducing the result modulo modulus */
template <typename typename_>
typename_ decrypt(typename_ encrypted_message,typename_ private_key, typename_ modul
us_)
{
    return modular_exponentiation(encrypted_message,private_key,modulus_);
}

/*this is a call function that opens a pipe to an external java application that uti
lizes
a built-in java function that verifies the primality of line2 with accuracy line3 */
template <typename typename_> int isprime(typename_ line2, int line3)
{
    FILE *fp;
    int status;
    char prime[WORD];
    string line1="java -classpath 3rd_party isProbablePrime ";
    stringstream line,totalline;
    totalline<<line1;
```

```
    totalline<<line2;
    totalline<<" ";
    totalline<<line3;
    string order = totalline.str();
    fp = popen(order.c_str(),"r");
    fgets(prime, WORD, fp);
    return (prime[0]-'0');
    status = pclose(fp);
}

/*this function decreases a given number (wanted) until that number is a prime*/
template <typename typename_> typename_ round_to_prime(typename_ wanted)
{
    while(!isprime(wanted,4))
        wanted--;
    return wanted;
}

/* a function that generates a proper public-key that satisfies the RSA constraint g
cd(public-key,phi(modulus))=1. */
template <typename typename_>
pub_key_sequence<typename_> generate_public_key(typename_ pub, typename_ p,typename_
 q)
{
    p=round_to_prime(p);
    q=round_to_prime(q);
    while((pub!=0)&&((!isprime(pub,4))||(gcd(pub,phi(p,q))!=1)))
        pub--;
    pub_key_sequence<typename_> result = {pub,p,q,(typename_)(p*q)};
    return result;
}

/* this is a wrapper function for the modular_inverse function that initializes the
previous with the proper paramiters */
template <typename typename_> typename_ calculate_private_key(typename_ public_key,t
ypename_ p,typename_ q)
{
    typename_ pub = public_key;
    typename_ temp_p = p;
    typename_ temp_q = q;
    typename_ private_key = modular_inverse(pub,temp_p,temp_q);
    return private_key;
}
#endif
```