```
#include <iostream>
#include <cstdlib>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <cstring>
#include <string>
#include <sstream>
#include <fstream>
#include <algorithm>
#include <string>
#include <vector>
#include <iterator>
#include <stack>
using namespace std;
/*********************************************************************************
* Applied Cryptography and Network Security - Final project  / Jack Shutzman     *
*                        NYU  G22.3205-001  11/20/2009         N15928458         *
**********************************************************************************
* 3 goals:  1) Building  RSA systems                                             *
*           2) Create a digital Certificate                                      *
*           3) Authenticate a user                                               *
*                                                                                *
* As part of those goals, I'll show traces of some processes that take place     *
* during the creation of the necessary elements to accomplish the goals. Also    *
* the program will demonstrate some aspects of the workings of the RSA Cryto-    *
* system, as required by the specifications of the project                       *
*                                                                                *
**********************************************************************************
*                                                                                *
* Some concepts and limitations                                                  *
*                                                                                *
* The program uses arrays of characters (or strings) to represent a sequence of  *
* bits. For example an array : char bits[32] will represent an integer in bits.  *
* Since we deal with relatively small numbers and limited arrays, it is not      *
* a real waste, although using 'real' bit arrays could have saved negligent      *
* amount of space, but this simulation with string simplifies the program.       *
* Of course all the values and the integers are kept as the actual values, so    *
* this internal simulation of bits with strings is transparent to the user.      *
* Accordingly I created my own version of bit manipulation, for instance rather  *
* then using C++ XOR function I wrote XORC (xor for character).                  *
* RANDOM function is used with a seed which is a system time to prevent repeat   *
* of the same result with every run. Still not completely random, but a tolerable*
```

```
* pseudo-random generation.                                                *
****************************************************************************/

//GLOBAL VARIABLES
int MAX_BITS = 32;
int MAX_SYSTEMS =2;
const int MAX_AS=20;            //MAXIMUM a's FOR PRIMALITY CHECK
int random_a_arr[MAX_AS];       //KEEPS RANDOM a'S GENERATED FOR MILLER-RABIN ALG.
int FIRST_E=5;                  //TO MAKE THINGS INTERESTING START FROM e=5
time_t current_seed=0;
bool trace_requested=false;
string mr_print_prime[35];      //STORING PRINTING LINE FOR PRIME/NONE PRIME CASES
string mr_print_none_prime[35];
bool random_printed_switch= false;
bool primality_y_switch=false;
bool primality_n_switch=false;
bool encryption_trace= false;
enum crypto_systems {Alice, Trent, Bob};  //FACILITATE LOOPING ON CRYPTO SYSTEMS
typedef enum crypto_systems crypto_systems;
inline crypto_systems& operator++(crypto_systems& rsa){ //OVERLOAD ++ TO LOOP
            return rsa = crypto_systems(rsa + 1);       //MORE ELEGANTLY
}
crypto_systems rsa;

string names[3]={"Alice", "Trent", "Bob"};
struct crypt{
             string name;
             int n;
                int public_key;
                int private_key;
                int p;
                int q;
} crypto[2];

struct cert{
             string name;
                int public_key;
                string r;
                int signature;
} Alice_cert;

struct euclid {
             int gcd;
```

```
                int multiplicative_inverse;
} euclid_result;

// FUNCTION PROTOTYPES (IN APHABETICAL ORDER)

int alice_decrypt_h_u(string);
int bit_to_num(string);
int bob_encrypt_v(int);
string bob_pick_u(int);
struct cert build_certificate(string, int, int);
int build_integer(void);
void build_RSA_systems(crypto_systems);
string char_to_bit(unsigned int);
string complete_byte(string);
void empty_print_mr(void);
int encrypt_decrypt(int,int, int);
struct euclid euclidean_extended(int, int);
string extract_virtual_byte(string, int);
int  extract_char(string, int);
int fast_exp(int, int, int);
int generate_none_prime(void);
int generate_prime(void);
int hash(string s);
string int_to_string(int);
bool miller_rabin(int, int);
int mod(int , int );
int new_a(int);
string num_to_bit(int);
string pad_str(string);
string pad_text(string, int);
bool prime(int);
void print_crypt(int);
void print_output_heading(void);
void print_prime(char);
void rnd_seed(time_t);
void space(int);
string xorb(string, string);
char xorc(char, char);


//END FUNCTION PROTOTYPES

int main (int argc, char * const argv[]) {
```

```
print_output_heading();    //REPORT START
   build_RSA_systems(Trent);  //CONSTRUCTS p, q, n, e AND d FOR PERSONS FROM
                           //ALICE UP TO TRENT (HAPPENS TO BE ONLY 2 PEOPLE)
print_crypt(Alice);        //PRINT ALICE'S CRYPTO SYSTEM IN INTEGER & BITS
                           //ENUMS IN C++ ARE EFFECTIVELY INTEGERS STARTING
                                 //FROM ZERO AS A DEFAULT

Alice_cert=build_certificate(" Alice", crypto[Alice].n,
                             crypto[Alice].public_key);

   //NOT REQUIRED, BUT COULD BE USED
   space(2);
   cout << "\nNOT REQUIRED PORTION";
   cout << "\n############################################################"
        << "#######";
   cout << "\n=======> Trent's keys: Public => (" << crypto[Trent].public_key
        << ", " << crypto[Trent].n << ")  Private => ("
        << crypto[Trent].private_key << ", " << crypto[Trent].n << ")";
   cout << "\n############################################################"
        << "#######";
   cout << "\nEND OF NOT REQUIRED PORTION";
   space(2);


//PRINT CERTIFICATE FOR ALICE
cout << "\n\n So here is Alice's certificate\n";
cout <<     "===============================";
cout << "\n Alice_cert.name= " << Alice_cert.name;
cout << "\n Alice_cert.e   = " << Alice_cert.public_key;
cout << "\n Alice_cert.r   = " << Alice_cert.r;
cout << "\n Alice_cert.s   = " << Alice_cert.signature;
   space(2);




   //BOB PICKS u
string u=bob_pick_u(crypto[Alice].n);
int u_int =bit_to_num(u);

cout << "\n So the u composed by Bob (according to the specs):   u[b]=" << u;
cout << "\n This u as integer is ==========================>     u   =   " << u_int;
```

```
cout << "\n\n So Bob sends to Alice u= " << u_int;

//ALICE AUTHENTICATES HERSELF
int v=alice_decrypt_h_u(u);
cout << "\n\n And then Alice [after hashing and decrypting with her "
     <<"private key] returns to Bob v= " << v;
   space(3);

//SHOW ENCRYPTION PROCESS
encryption_trace=true;
cout << "\n\n Showing Bob's encryption of V  E(V, e)";
cout << "\n========================================";
space(2);

//BOB VERIFIES ALICE'S IDENTITY
int z= bob_encrypt_v(v);
int h_bob=hash(u);

cout << "\n\n Bob encrypts v with Alice's public key and gets Z= " << z;
cout << "\n Bob also does hash of: " << u_int << " which in binary is: ";
cout << u << "  and he gets h(u)= " << h_bob;
space(2);

cout << "\n\n AND IF THEY MATCH [Z == h(u)], BOB KNOWS HE IS TALKING TO";
cout << "\n SOMEONE WHO'S GOT ALICE'S PRIVATE KEY (HOPEFULLY ALICE)\n";
space(2);
//PRINT SUMMARY OF VALUES FOUND
cout  <<  "\n\n TO SUMMARIZE"
      <<  "\n ------------"
      <<  "\n\n"
      <<  " In Integers: \n\n"
      <<  "           u           = " << u_int
      <<  "\n           h(u)        = " << h_bob
      <<  "\n           V=D(d,h(u)) = " << v
      <<  "\n           E(e,V)      = " << z
      <<  "\n\n In Binary: \n\n"
      <<  "           u           = " << u
      <<  "\n           h(u)        = " << pad_str(num_to_bit(h_bob))
      <<  "\n           V=D(d,h(u)) = " << pad_str(num_to_bit(v))
      <<  "\n           E(e,V)      = " << pad_str(num_to_bit(z))
      <<  "\n\n";
```

```
        return 0;
}


/******************************  END OF MAIN PROGRAM ****************************/

/*******************      FUNCTIONS SORTED ALPHABETICALLY     *******************/


int alice_decrypt_h_u(string in){ //ALICE AUTHENTICATES HERSELF TO BOB
    int h=hash(in);
    return encrypt_decrypt(h,crypto[Alice].private_key, crypto[Alice].n);
}
/********************************************************************************/

int bit_to_num( string bit_str){ // CONVERTING A BIT STRING INTO AN INTEGER

        int len=bit_str.size()-1;
     double result=0;
     for(int i=len; i>=0; --i){
         if(bit_str[i]=='1'){
             result += pow(2, len-i);
         }
     }
     int res=static_cast<int>(result);
     return res;
}
/********************************************************************************/
int bob_encrypt_v(int v){
    return encrypt_decrypt(v,crypto[Alice].public_key, crypto[Alice].n);

}
/********************************************************************************/

string bob_pick_u(int n){
/*********************************************************************************
* BOB USES ALICE'S CRYTO SYSTEM ACCORDING TO ALICE'S n AND THE SPECIFICATIONS OF *
* THE k AND THE RANDOM NUMBER GENERATION ROUTINE. SHIFTING THE FIRST '1' FOUND   *
* IN (BINARY) n ONE BYTE TO THE RIGHT (AS DESCRIBED IN THE SPECS) GUARANTEES     *
* TO PRODUCE u < n                                                               *
*********************************************************************************/
         string bits_n = pad_str(num_to_bit(crypto[Alice].n));
```

```
        cout << "\n Alice's 'n' in 32 bits is:" << bits_n;
        int k=MAX_BITS -1-bits_n.find_first_of('1');
           cout
        << "\n\n K ===> Bob found first '1' from left of 'n' in position ===> K= "
        << k;
        space(2);
        string u_string="";
        for(int i=0; i<MAX_BITS -k; ++i){
            u_string +='0';
        }
        u_string +='1';
        for(int j=0; j<k-1; ++j){
             //RANDOMLY GENERATE THE NEXT k-1 BITS
             int gen_rand = rand();
             string bits=num_to_bit(gen_rand);
             u_string +=bits[bits.size()-1]; //LAST BIT OF THE RANDOMLY GEN. INTEGER
        }
        return u_string;
}




/*******************************************************************************/

struct cert build_certificate(string name, int n, int pub_key){
/*******************************************************************************
* BUILDING THE DIGITAL CERTIFICATE IS CONCATENATING THE NAME AND THE           *
* PUBLIC KEY [(n,e) PAIR], HASHING THE RESULTING STRING AND DECRYPTING         *
* THE RESULT WITH TRENT'S (AUTHORITY) PRIVATE KEY. EVERYONE KNOWS TRENT'S      *
* PUBLIC KEY AND HE IS TRUST-WORTHY (SO WE WERE TOLD)                          *
* WHEN DEALING WITH THE NAME AND CHARACTERS THAT ARE NOT NUMERIC, I FIRST      *
* CONVERT THEM TO THEIR ASCII REPRESENTATION AND PROCEED WITH THAT VALUE,      *
* FOR INSTANCE 'A' WILL BE 65 (DEC) OR 01000001(binary)                        *
* AS I MENTIONED BEFORE, A BYTE IS REPRESENTED BY 8 CHARACTERS OF '1'S AND '0'S *
*******************************************************************************/
          string r_pair="";
       struct cert al;
       //FIRST ATTACH THE NAME TO THE STRING OF THE PAIR r
       int len=name.size();
       for (int i=0; i<len; ++i){                      //FIRST 6 BYTES OF r
           int ch=extract_char(name,i);                //TURN CHAR TO ASCII VALUE
```

```
            string byte1=complete_byte(num_to_bit(ch));    //FULL BYTE=8 BITS
            r_pair += byte1;
        }
           r_pair +=pad_str(num_to_bit(n));                //PADDED n BYTES  7-10
        r_pair +=pad_str(num_to_bit(pub_key));          //PADDED e BYTES 11-14

        //FIRST HASH r_pair ( NAME + PUBLIC KEY )
        int h=hash(r_pair);

        //TRENT'S SIGNS BY DECRYPTING THE HASHED VALUE h, WITH HIS PRIVATE KEY
        int s=encrypt_decrypt(h,crypto[Trent].private_key, crypto[Trent].n);

        //DISPLAYING THE RESULTS OF: r_pair, THE HASH, AND SIGNATURE.

        cout << "\n\n Here are: r, h(r) and s as bits \n";
        cout <<     " -----------------------------\n";
        cout << "\n r   =" << r_pair << "\n";
        cout << " h(r)=" << pad_str(num_to_bit(h)) << "\n s   ="
             << pad_str(num_to_bit(s));
        space(1);
        cout << "\n\n Here are: h(r) and s as integers \n";
        cout <<     " ------------------------------\n";
        cout << " h(r)=" << h << "\n s   =" << s;
        space(2);
        //ASSIGN ALICE'S CERTIFICATE ELEMENTS
        al.name=name;
        al.r = r_pair;
        al.public_key = pub_key;
        al.signature  = s;

        return al;
}
/*******************************************************************************/

int build_integer(void){ //GENERATING A RANDOM INTEGER< 128 IN BINARY < 10000000
//CANDIDATE FOR p, q -> EACH BINARY DIGIT (1-5) IS DECIDED BY DIFFERENT RANDOM

      string seven_i="1000001"; //AS REQUIRED BY THE SPECS BITS 1 AND 7 FIXED '1'
      if (!random_printed_switch)
         cout << "\n\n TRACE OF GENERATING SAMPLE INTEGER"
              << "\n ==================================";
     rnd_seed(++current_seed);
     for(int i=0; i<5; ++i){       //BITS 1-5 ARE ASSIGNED RANDOMLY
```

```
        int gen_r = rand();
        string bits=num_to_bit(gen_r);
        seven_i[i+1]=bits[bits.size()-1];
        //REQUIRED OUTPUT
        if (!random_printed_switch){
             cout << "\n random number  : " << i << " is: " << gen_r;
          cout << "\n In binary it is:       " << bits;
             cout << "\n The least significant bit of that number is obviously ==> "
               << bits[bits.size()-1];
          cout << "\n So therefore bit number " << i+1 << "  from left is: "
                  << "                 " <<bits[bits.size()-1] << "\n";
        }

     }
     space(2);
     string int_32=pad_str(seven_i);

     if (!random_printed_switch){
           cout << "\n 32-bit Padded random integer: " << int_32 << "\n";
           space(2);
           random_printed_switch=true;
     }
     return bit_to_num(int_32);
}
/********************************************************************************/

void build_RSA_systems(crypto_systems limit){
/********************************************************************************
* CREATING THE CRYPTO SYSTEM ELEMENTS FOR BOTH ALICE AND TRENT, ENSURING THAT   *
* p, q ARE DIFFERENT WITHIN EACH CRYPTO SYSTEM AND ENSURING THAT p TRENT AND q  *
* TRENT ARE DIFFERENT THAN p ALICE AND q ALICE (SO BY DEFINITION n OF ALICE IS  *
* DIFFERENT THAN n OF TRENT).                                                   *
* THE SELECTION OF THE TRYS FOR A PUBLIC KEY START FROM 5 FOR ALICE, AND STARTS *
* FROM ONE MORE THAN WHAT HAD BEEN FOUND APPROPRIATE FOR ALICE, ENSURING THE    *
* PUBLIC KEYS FOR ALICE AND TRENT ARE DIFFERENT.                                *
********************************************************************************/


     int e=FIRST_E;          //STARTING TO CHECK POSSIBLE PUBLIC KEYS
     int p; int q;
     empty_print_mr();
        for(rsa=Alice; rsa<= limit; ++rsa){
         //FIRST SYSTEM IS FOR ALICE, SECOND FOR TRENT
```

```
if(!primality_n_switch){
      mr_print_none_prime[1]=
      "   i    X[i]        z               y        y";
      mr_print_none_prime[2]=
      " -----------------------------------------------";

      p=generate_none_prime(); //FOR ILLUSTRATING NONE PRIME DETECTION
      print_prime('n');        //'n' STANDS FOR NONE PRIME
      cout << "\n The integer: " << p << "  is not a prime!" <<"\n\n";
      primality_n_switch=true;
}

p=generate_prime();
if(!primality_y_switch){     //PRINTING ONE CASE OF 'PERHAPS PRIME'
      mr_print_prime[1]     =
      "   i    X[i]        z               y        y";
      mr_print_prime[2]     =
      " -----------------------------------------------";
      print_prime('y');       //'y' STANDS FOR YES PRIME
      cout << "\n The integer: " << p << "  is perhaps a prime!" <<"\n\n";
      primality_y_switch=true;
}


if(rsa==Trent){ //ENSURING TRENT'S p IS DIFFERENT THAN ALICE'S p AND q
    while(p==crypto[Alice].p || p==crypto[Alice].q){
         p=generate_prime();
    }
}
 while((q=generate_prime())==p); //ENSURING p != q WITHIN A CRYPTO SYSTEM

 int n=p*q;
 int phi_n=(p-1)*(q-1);

 //SELECTING e PROCESS, WHEN FOUND, CALCULATE d with X-EUCLIDEAN ALG.

 if (rsa==Trent) ++e;           //AVOID DUPLICATING THE PUBLIC KEY
 bool suitable_e=false;
 //SEARCH LOOP FOR APPROPRIATE e
 trace_requested=true;          //TRIGGERS PRINTOUT OF ALICE'S e SEARCH
 if(trace_requested && rsa==Alice)
    cout << "\n\n\n\n  Our n=" << n << " Our Phi(" << n << ")="
```

```
                << phi_n << "\n\n";
        while(!suitable_e){
                euclid_result = euclidean_extended(phi_n,e);
             if (euclid_result.gcd==1)  //THEN e  AND phi_n ARE RELATIVELY PRIME
                 suitable_e=true;       //END THE SEARCH LOOP, OTHERWISE CHECK
             else                       //ANOTHER e
                 ++e;
        }
        //ASSIGN THE CRYPTO SYSTEM ELEMENTS TO THE STRUCT OF THE PROPER RECIPIENT
        crypto[rsa].name          =names[rsa];
        crypto[rsa].public_key    =e;
        crypto[rsa].private_key   =euclid_result.multiplicative_inverse;
        crypto[rsa].n             =n;
        crypto[rsa].p             =p;
        crypto[rsa].q             =q;


     }
}
/*********************************************************************************/

string char_to_bit(unsigned int u){
//CONVERT ONE CHARACTER INTO 8-BIT STRING. THE INPUT IS AN UNSIGNED INTEGER THAT
//REPRESENTS A CHARACTER.
       int t;
       string s1="";
       for(t=128; t>0; t = t/2)
         if(u & t) s1 += '1';
         else      s1 += '0';
       return s1;
}
/*********************************************************************************/

string complete_byte(string in_ch){
// COMPLEMENTS TO A FULL BYTE (LEADING ZEROS) - VIRTUAL BYTE REPRESENTED BY CHARS.
// THIS PROCESS IS NEEDED WHEN WE DEAL WITH A WHOLE BYTE, WHICH IS 8-BITS
// WHEN THE ASCII CODE OF A CHARACTER DOES NOT FILL 8-BITS, WE COMPLEMENT IT HERE
       int len_remain=8-in_ch.size();
       string res=in_ch;
       while(len_remain >0){
           res='0'+res;
        --len_remain;
       }
```

```
       return res;
}


/*********************************************************************************/

void empty_print_mr(void){ //INIT ARRAYS USED TO ACCUMULATE PRINT ROWS
    for(int i=0; i<35; ++i){
        mr_print_prime[i]="";
        mr_print_none_prime[i]="";
    }
}
/*********************************************************************************/

int encrypt_decrypt(int h,int private_key, int n){ //ENCRYPT OR DECRYPT

      return fast_exp(n,h,private_key); //USING FAST EXPONENTIATION TECHNIQUE
}
/*********************************************************************************/

struct euclid euclidean_extended(int phi,int e){
/********************************************************************************
* IMPLEMENTATION OF THE EXTENDED EUCLIDEAN ALGORITHM TOGETHER WITH PRINTING     *
* THE TABLE OF VALUES PROGRESSION FOR THE RELEVANT VARIABLES, AS DONE IN CLASS. *
* THIS IS ONE OF THE MAJOR ELEMENTS OF AN RSA CRYPTO SYSTEM - FINDING A PUBLIC  *
* KEY AND ITS MULTIPLICATIVE INVERSE (WHICH WILL SERVE AS THE PRIVATE KEY.      *
* THIS FUNCTION ALSO ILLUSTRATES THE FINDING OF AN APPROPRIATE e. IT STARTS     *
* FROM AN ARBITRARY VALUE OF 5 (MY CHOICE) AND IF 5 IS NO GOOD, IT ALSO SHOWS   *
* THE PROCESS OF FAILURE WITH THAT CHOICE.                                      *
********************************************************************************/
       struct euclid eu;
       int r, r1, r2, s, s1, s2, t, t1, t2, q;

       r1=phi; r2=e;
       s1=1; s2=0;
       t1=0; t2=1;
       if(trace_requested && rsa==Alice){
          cout << "\nChecking if e=" << e << " can be a public key"
                << " [Extended Euclidean Algorithm]\n";
          cout << "\n    q    r1    r2    r    s1    s2    s    t1    t2     t \n"
                << "    -------------------------------------------------------\n";
       }
       while(r2 > 0){
```

```
            q=r1/r2;
             r = r1 - q*r2;
             s = s1 - q*s2;
             t = t1 - q*t2;
             if (trace_requested && rsa==Alice)
                 printf("%5d %5d %5d %5d %5d %5d %5d %5d %5d %5d\n", q, r1 ,   r2 , r,
                         s1,    s2,    s,    t1,    t2,    t);

             r1=r2; r2=r;
             s1=s2; s2=s;
             t1=t2; t2=t;

         }
         eu.gcd=r1; s=s1; t=t1;
            if(eu.gcd==1){
                eu.multiplicative_inverse= (t<0) ? t+phi:t;
             if (trace_requested && rsa==Alice){
                 cout << "\n\n FOUND IT !  e will be:" << e
                      << "  Its multiplicative inverse in Z(" << phi << ") is:"
                   << eu.multiplicative_inverse << "\n So, ==> d="
                   << eu.multiplicative_inverse;
                   if (e==FIRST_E) cout << "\n We Got Lucky On The First Try";
                   space(1);
             }
         }
         else{
             if (trace_requested && rsa==Alice)
                  cout << "\n Unfortunately e=" << e
                       << " Does not have a multiplicative inverse in Z("
                       << phi << "), so we go ahead and try the next e\n\n";
         }

         return eu;
}




/*********************************************************************************/

string extract_virtual_byte(string in, int i){
/*********************************************************************************
```

```
* EXTRACTING 8 CHARACTERS FROM THE INPUT STRING STARTING AT LOCATION SPECIFIED *
* BY i. THOSE 8 CHARACTERS REPRESENT A VIRTUAL BYTE. THE PURPOSE OF THIS FUNCTION*
* IS TO FACILITATE SEPARATING A STRING INTO 'BYTES' FOR THE HASH FUNCTION.       *
*********************************************************************************/

        string out="";
        int loc=i*8;
        for(int k=loc; k<loc+8; ++k){
            out += in[k];
        }
        return out;
}
/********************************************************************************/

int  extract_char(string s, int i){
// EXTRACTS CHARCTER SEQUENCE i OUT OF STRING S COUNTING FROM LEFT, FIRST ONE
// INDEXED ZERO - AND RETURNS ITS ASCII VALUE AS INTEGER
     char byte=s[i];
     int asciival=static_cast<int>(byte);
     return asciival;
}
/********************************************************************************/

int fast_exp(int n, int a, int x){ //IMPLEMENT FAST EXPONENTIATION  a^x(mod n)
// n - MODULUS,  a - INTEGER BASE,  x - EXPONENT IN DECIMAL NUMBER

   int prev_y=0;
   if(encryption_trace){ //PRINT THE TABLE OF PROGRESSION AS DONE IN CLASS NOTES
       cout <<
               "\n                           Squaring                      "
            << "     Multiplying";
       cout << "\n i    X[i]                          Y"
            << "                                        Y    ";
       cout << "\n---------------------------------------"
            << "-------------------------------------------";
   }

    string xbits=num_to_bit(x); // FIRST CREATE THE BIT STRING REPRESENTING x
     int k=xbits.size();
     int y=1;
     for(int i=0; i<k; ++i){
        if(encryption_trace)
           cout << "\n" << k-i << "      " <<  xbits[i];
```

```
        prev_y=y;
        y=mod(y*y,n);

        if(encryption_trace){
            cout << "       ";
            string temp1= int_to_string(prev_y) + "^2(mod " + int_to_string(n) +
                          ")=" + int_to_string(y);
            string sqr=pad_text(temp1,30);
            cout << sqr;
        }
        if(xbits[i] == '1'){
           int temp_y=y;
           y=mod(a*y,n);
           if(encryption_trace){
                string temp2= int_to_string(a) + "x" + int_to_string(temp_y) +
                              "(mod " + int_to_string(n) + ")=" +
                             int_to_string(y);
                string mul=pad_text(temp2,30);
                cout << "     " <<  mul;
           }
        }
        else{
             if(encryption_trace){
                cout << "     " << pad_text(int_to_string(y),30);
            }
        }
     }
     if(encryption_trace)
        cout << "\n-------------------------------------"
             << "-----------------------------------------";
     return y;
}
/*********************************************************************************/

int generate_none_prime(void){
//FOR DEMONSTRATION OF NONE-PRIME FOUND ONLY. build_integer() IS CALLED TWICE IN
//ORDER TO VARY INTEGERS BETWEEN RUNS.
    while(1){
           int none_prime=build_integer();
           none_prime=build_integer();  //SECOND CALL ENSURES DIFFERENT RESULT
           if(!prime(none_prime)) return none_prime;

    }
```

```
}
/*********************************************************************************/
int generate_prime(void){
//LOOP UNTIL A PRIME'S FOUND (WITH HIGH PROBABILITY).
    while(1){
          int prime_candidate=build_integer();
          if(prime(prime_candidate)) return prime_candidate;
    }
}
/*********************************************************************************/
int hash(string r){ //ONE WAY HASH FUNCTION (h)
/********************************************************************************
* THIS FUNCTION RECEIVES THE STRING TO BE HASHED. THE STRING IS ALREADY        *
* EXPANDED TO BITS, SO FOR EXAMPLE THE 14 BYTES OF THE CERTIFICATE WILL BE     *
* PLACED PRIOR TO THIS HASH IN A STRING OF 8X14=112 CHARACTERS.                *
* ACCORDING TO THE SPECS, THIS ROUTINE SEPARATES THE INPUT INTO 'BYTES', EACH  *
* BYTE IS 8 BITS AND IT CALLS xorb, IN A LOOP TO XOR ALL INPUT BYTES AND       *
* GENERATE AN OUTPUT WHICH IS A STRING OF 8 BITS. THAT OUTPUT IS THEN TURNED   *
* INTO AN INTEGER BY PERFORMING CONVERSION FROM BIT TO NUM AND PADDING WITH    *
* ZEROS TO OBTAIN A FINAL OUTPUT OF AN INTEGER OF 32 BITS. THIS FUNCTION WORKS *
* ON INPUT OF ANY LENGTH AND ALWAYS RETURNS AN INTEGER VALUE.                  *
********************************************************************************/

   string temp;
   int len=r.size()/8;
   string ch=extract_virtual_byte(r,0);   //STORE FIRST BYTE
   for(int k=1; k<len; ++k){
      temp=extract_virtual_byte(r,k);
      ch=xorb(ch,temp);                   //XOR WITH FOLLOWING BYTES
   }
   int result=bit_to_num(pad_str(ch));
   return result;

}
/*********************************************************************************/
string int_to_string(int i){
/********************************************************************************
* TURN AN INTEGER INTO A STRING AND PAD IT WITH SPACES FROM LEFT FOR PRINTING.  *
* THIS FUNCTION ALLIGNS AN INTEGER TO THE RIGHT OF A 5 CHARACTER STRING AND PADS*
* IT WITH SPACES ON THE LEFT. SINCE n CANNOT EXCEED 5 DIGITS (<127^2), SIZE OF  *
* 5 IS SUFFICIENT, I.E.   123 WILL RETURN "  123",  4356 WILL RETURN: " 4356"   *
* THIS FORMATING WILL RESULT IN UNIFORM PRINT INTO AN ORGANIZED ALLIGNED TABLE  *
* WE ALSO TAKE 'MODULO' n AFTER EVERY OPERATION TO ENSURE WE WORK WITH NUMBERS  *
```

```
* SMALLER THAN n.                                                              *
*******************************************************************************/

    stringstream temp;
    temp << i;
    string padded= "";
    string actual_n=temp.str();      //TURN THE INTEGER INTO A STRING
    padded=pad_text(actual_n,5);
    return padded;

}
/*******************************************************************************/
bool miller_rabin(int candidate, int a){
/*******************************************************************************
*  CHECK IF AN INTEGER IS A PRIME NUMBER PROBABLISTIC ALGORITHM. THIS          *
*  IMPLEMENTATION OF THE MILLER-RABIN ALGORITHMS IS ONE OF THE MAJOR BUILDING   *
*  BLOCKS OF OUR CRYPTO SYSTEM.                                                *
*  THE ARRAYS mr_print_prime AND mr_print_none_prime SAVE THE RESULTS FOR PRINT *
*******************************************************************************/
     string biti;
     int      dec_exponent=candidate-1;
     string   bin_exponent=num_to_bit(dec_exponent);
     int y=1;
     int z;

     for(int i=0; i< bin_exponent.size(); ++i){    //k STARTS FROM THE LEFT OF
              z=y;                                  //THE BIT STRING
           (bin_exponent[i]=='0')? biti='0' : biti='1';

           if(!primality_y_switch){
              mr_print_prime[i+3]="";
              mr_print_prime[i+3] += int_to_string(bin_exponent.size()-i-1)+ "     ";
              mr_print_prime[i+3] += biti + "       ";
              mr_print_prime[i+3] += int_to_string(z) + "         ";
           }
           if(!primality_n_switch){
              mr_print_none_prime[i+3]="";
              mr_print_none_prime[i+3] += int_to_string(bin_exponent.size()-i-1)+
                                          "     ";
              mr_print_none_prime[i+3] += biti  + "       ";
              mr_print_none_prime[i+3] += int_to_string(z) + "         ";
           }
```

```
            y *=y;
            y=mod(y, candidate);

            if(!primality_y_switch)
               mr_print_prime[i+3] += int_to_string(y) + "     ";
            if(!primality_n_switch)
               mr_print_none_prime[i+3] += int_to_string(y) + "     ";

            if (y==1 && z != 1 && z!= (candidate-1)){
                mr_print_none_prime[0]="Checking the integer: " + int_to_string(candidate)
+
                             " , Using a= " +  int_to_string(a);
                return false; //MILLER-RABIN FOUND OUT CANDIDATE SURELY NOT PRIME
             }
            if(bin_exponent[i] == '1'){
                  y *=a;
                  y= mod(y,candidate);
            }

            if(!primality_y_switch)
               mr_print_prime[i+3] += int_to_string(y);
            if(!primality_n_switch)
               mr_print_none_prime[i+3] += int_to_string(y);


      }
      if(y !=1){
         mr_print_none_prime[0]=" Checking the integer: "+int_to_string(candidate)+
                             " , Using a= " +  int_to_string(a);
         return false; //MILLER-RABIN FOUND OUT CANDIDATE SURELY NOT PRIME
      }
      else {
         mr_print_prime[0] =" Checking the integer: " + int_to_string(candidate) +
                             " , Using a= " +  int_to_string(a);
         return true; //MILLER-RABIN FOUND OUT CANDIDATE IS PERHAPS PRIME
      }

}
/********************************************************************************/
int mod(int m, int n){ //RETURNS m(mod n) - MY OWN MOD FUNCTION
      return m - (m/n)*n;
}
```

```
/******************************************************************************/
```

```
int new_a(int n){
/*****************************************************************************
* TAKES n AS INPUT AND RETURNS A RANDOM NUMBER a, SUCH THAT 0 < a < n. PRODUCES *
* DIFFERENT RANDOM a'S UP TO MAX_AS (20) WHEN CALLED REPEATEDLY FOR THE         *
* MILLER-RABIN PRIMALITY CHECK (USED BY PRIME FUNCTION). I COULD HAVE MADE THIS *
* FUNCTION SLIGHTLY MORE EFFICIENT BY PASSING A PARAMETER OF HOW MANY a's WERE  *
* FOUND SO FAR, BUT SINCE IT IS ONLY UP TO 20, I DECIDED TO LIVE WITH THIS      *
* INEFFICIENCY AS A TRADE-OFF TO SIMPLIFYING THE CODE.                          *
*****************************************************************************/
    int res=mod(rand(),n);
    bool new_a_found=false;
    while(!new_a_found){
         new_a_found=true;
         for (int j=0; j<MAX_AS; ++j) //ENSURES a != PREVIOUSLY FOUND a'S
           if (res==random_a_arr[j] || res==0)
                 new_a_found=false;
           if(new_a_found) return res;
           else res=mod(rand(),n);
    }
}
/******************************************************************************/
string num_to_bit(int number){ // CONVERT AN INTEGER TO A STRING OF ITS BITS VALUE
     int quotient=number;      // USING ELEMNTARY MATH (DEC TO BINARY CONVERSION)
     string result;
     int modulus;
     while(quotient > 0){
          modulus = mod(quotient, 2);
          quotient /=2;
          result += (char) (modulus + '0');  // CONVERTING THE MODULUS INTO A CHAR
     }
     reverse(result.begin(), result.end());
```

```
     return result;
}




/*****************************************************************************************/
string pad_str(string st1){ //PADDING A STRING WITH ZEROS ON LEFT TO MAX_BITS
   int len=st1.size()-1;
   string padded= "";
   for(int j=0; j<MAX_BITS; ++j){
       padded +='0';
   }
   for(int i=0; i<=len; ++i){
       padded[MAX_BITS-i-1]=st1[len-i];
   }
   return padded;
}
/*****************************************************************************************/
string pad_text(string txt, int size){
/****************************************************************************************
* THE FIRST INPUT PARAMETER IS THE STRING TO BE ALLIGNED. THE SECOND DETERMINES *
* THE SIZE OF THE OUTPUT. IF THE OUTPUT SIZE IS LONGER THAN THE INPUT STRING    *
* (NORMAL CASE), THE OUTPUT IS PADDED WITH BLANKS FROM LEFT, IF SMALLER, THEN   *
* THE INPUT IS RETURNED UNCHANGED.                                              *
****************************************************************************************/
         string pattern;
      int len=txt.size();
      if(len >= size) return txt;
      for (int i=0;i<size-len; ++i){
          pattern += ' ';
      }
      for(int j=0; j<len; ++j){
          pattern += txt[j];
      }
      return pattern;

}
/****************************************************************************************/
```

```
bool prime(int cand){
/*******************************************************************************
* THIS IS ANOTHER IMPORTANT FUNCTION. IT USES RABIN-MILLER SEVERAL TIME ( UP TO *
* MAX_AS) TO ENSURE HIGH CONFIDENCE IN OUR PRIMALITY CHECK. IT RECEIVES AS INPUT*
* AN INTEGER AND RETURNS 'TRUE' IF THE INTEGER IS PRIME AND 'FALSE' OTHERWISE.  *
* THE FALSE INDICATES 'DEFINITELY NOT PRIME', THE TRUE INDICATES PRIME WITH     *
* PROBABILITY OF : 1 - 1/2^MAX_AS ACCORDING TO THEOREM 14 IN THE CLASS NOTES    *
*******************************************************************************/

     int a_counter=1;
     current_seed++;
     rnd_seed(current_seed);       //SEED THE RANDOMIZER WITH THE SYSTEM'S TIME
     for(int i=0;i<MAX_AS;++i)
         random_a_arr[i]=0;
        int a=new_a(cand);            //RANDOMLY PICKED a  SUCH THAT  0 < a < cand
     bool prime_f=true;
     while (prime_f && a_counter < MAX_AS){
          if (!miller_rabin(cand,a)){
               prime_f=false;
          }
          else{
               random_a_arr[a_counter++]=a;
               a=new_a(cand);
          }
     }
     return prime_f;
}
/********************************************************************************/
void print_crypt(int a){//PRINTING THE CRYPTO SYSTEM GENERATED FOR A RECEPIENT
          cout << "\n HERE IS " << names[a] << "'S CRYPTO SYSTEM";
       cout << "\n ------------------------------";
       cout <<"\n As Integers:";
       cout << "\n               p = " << crypto[a].p;
       cout << "\n               q = " << crypto[a].q;
       cout << "\n               n = " << crypto[a].n;
       cout << "\n               e = " << crypto[a].public_key;
       cout << "\n               d = " << crypto[a].private_key;
       cout << "\n";
       cout <<"\n As BIT strings:";
       cout << "\n               p = " << num_to_bit(crypto[a].p);
       cout << "\n               q = " << num_to_bit(crypto[a].q);
       cout << "\n               n = " << num_to_bit(crypto[a].n);
```

```
       cout << "\n               e = " << num_to_bit(crypto[a].public_key);
       cout << "\n               d = " << num_to_bit(crypto[a].private_key);
       space(1);
}
/********************************************************************************/
void print_output_heading(void){
        time_t curr=time(0);
     char *t=ctime(&curr);
     cout <<
     "\n\n\nRSA program Report / Started running on: " << t<<"\n"
     <<"*****************************************************************\n\n"
     <<"*****************************************************************\n"
     <<"*****************************************************************\n"
     <<"*****************************************************************\n"
     <<    "\n\n\n";
}
/********************************************************************************/
void print_prime(char c){//PRINTS PRIME AND NONE PRIME AS SAMPLES FOR THE PROCESS
     if(c=='y'){
        cout << "\n THIS IS A SAMPLE FOR A SUCCESSFUL PRIMALITY CHECK "
             << "(PERHAPS PRIME)";
        cout    << "\n ====================================================="
                << "============";
        space(1);
        cout << "\n" << mr_print_prime[0];
        space(1);
     }
     else {
        cout << "\n THIS IS A SAMPLE FOR AN UNSUCCESSFUL PRIMALITY CHECK";
        cout    << "\n ====================================================";
        space(1);
           cout << "\n" << mr_print_none_prime[0];
        space(1);
     }
     int i=1;
     if(c=='y'){
        while(mr_print_prime[i] != ""){
             cout << "\n" << mr_print_prime[i];
          ++i;
        }
        cout << "\n -----------------------------------------------";
     }
     else{
```

```
        while(mr_print_none_prime[i] != ""){
             cout << "\n" << mr_print_none_prime[i];
          ++i;
        }
        cout << "\n ---------------------------------------------";
     }
}
/*********************************************************************************/
void rnd_seed(time_t s){//GENERATE A RANDOM NUMBER WITH C++ SEEDED RANDOM FUNCTION
                            //USE SYSTEM TIME TO VARY RANDOMS FOR EVERY RUN
     time_t seconds;
     seconds=time(NULL);
     seconds +=s;
     srand ( seconds );
}
/*********************************************************************************/
void space(int lines){ //LINE SPACE. USED TO MAKE THE REPORT PRINTED MORE LEGIBLE
   for(int j=0; j<lines; ++j){
      cout << "\n";
   }
}
/*********************************************************************************/
string xorb(string a1, string a2){ //THIS FUNCTION XORs A FULL BYTE WITH ANOTHER
//EACH BYTE IS REPRESENTED AS A STRING OF 8 CHARACTERS, EACH OF WHICH IS '0' OR '1'
        string result=""; char ch;
     int len=a1.size();
     for(int i=0;i<len;++i){
        ch=xorc(a1[i],a2[i]);
        result +=ch;
     }
     return result; //RETURNS ONE 'BYTE' REPRESENTED AS A STRING OF 8 '1'S & '0'S
}
/*********************************************************************************/
char xorc(char bit1, char bit2){ //SIMILAR TO C++'S XOR, BUT FOR BITS (REPRESENTED
                                 //IN CHARACTERS)
        if(bit1==bit2)
         return '0';
     else
         return '1';
}
/*********************************************************************************/
```