ÁLGEBRA ABSTRACTA

INFORME FINAL DEL RSA

- SEBASTIAN POSTIGO AVALOS (100%)
- JOAQUIN CASUSOL ESCALANTE (100%)
- FABIAN CONCHA SIFUENTES (100%)
- SEBASTIAN TINAJEROS ESTRADA (100%)
- FRANK ROGER SALAS TICONA (100%)

SEMESTRE 3

2021

“El alumno declara haber realizado el presente trabajo de acuerdo a las normas de la Universidad Católica San Pablo”

Abstract

Haremos un informe final detallado del RSA, en el que usaremos las funciones que investigamos en anteriores informes para poder crear un RSA lo más optimizado posible.

1. Estructura del main()

Primero declaramos nuestras variables iniciales:

Variables:

De tipo zz:

bits: De cuantos bits será nuestro RSA:
p, q, e: Para generar un RSA con claves predeterminadas (para cifrar o descifrar).
ea, Na: Las claves públicas del emisor.
eb, Nb: Las claves públicas del receptor.

De tipo bool:

WithSig: Para saber si el mensaje tiene firma o no.

Vectores de tipo string:

Cypher y DCypher: Para guardar los bloques de cifrado y descifrado.

De tipo string:

alpha: El abecedario que usaremos en el RSA.
sig: La firma que usaremos en el RSA.
message: El mensaje que ciframos.

Luego declaramos un char llamado input, usaremos input para escoger entre diferentes opciones del programa, la primera siendo TEST COMPLETO: Que creará un emisor y receptor y se cifrara/descifrara uno a otro. Generar números: Que creará una instancia de la clase RSA y creará números pseudo-aleatorios. Cifrar o descifrar: que como dice el nombre, se encarga de cifrar o descifrar con claves predeterminadas.

TEST COMPLETO:
Hacemos una llamada a nuestra clase RSA, e instanciamos dos objetos llamados Emisor y Receptor, a los cuales les pasamos los parámetros correspondientes, los cuales son nuestras variables bits, alpha y firma, mencionadas anteriormente.

Con respecto a nuestras claves públicas, declaramos las siguientes variables ZZ:

```
ea = Emisor.e;
Na = Emisor.N;
eb = Receptor.e;
Nb = Receptor.N;
```

* Donde “a” representa al emisor y “b” al receptor

Estas variables serán las claves que usaremos para el cifrado y descifrado, por lo que nosotros las podemos alterar según sea necesario, para poder usar las claves de otros usuarios.

Además, tenemos una variable booleana llamada WithSig, la cual nos indicará si nuestro programa hará uso de nuestra firma digital, o no.

Referente al cifrado del emisor, Se hará uso de un vector, de tipo string, será llamado Cypher, y será el cifrado con firma que el emisor realizará, llamando a la función del Emisor **cypherwithsignature**, se le pasará como parámetros el mensaje que se quiere cifrar, las

claves públicas eb y Nb. Como resultado se imprimirá el mensaje cifrado y su respectiva firma cifrada, por separado.

Por otro lado, el descifrado del receptor, hará uso de un vector de tipo string, y será llamado Dcypher, el cual contendrá el mensaje descifrado, por lo que se hace llamado a la función del Receptor **decypherwithsignature**, se le pasará como parámetros el mensaje cifrado anteriormente y las claves públicas ea y Na. Finalmente, se imprimirá el mensaje descifrado y su firma descifrada, por separado.

Generar numeros:
Creará una instancia del RSA llamada random usando un constructor, lo que generará las claves del RSA.

Cifrar o descifrar:
Haremos uso de nuestras variables: p,q y e. Con las que crearemos una instancia del RSA llamada testing usando el constructor que nos pide como parámetro las las variables mencionadas, luego le daremos valor a nuestras variables eb, Nb, que son las claves públicas del receptor. Ahora usaremos input para decidir entre cifrar o descifrar, si ciframos: igualamos Cypher a testing.cypherwithsignature, que usará como parámetros las claves públicas del receptor y message. Si desciframos: Haremos 2 pushback a Dcypher, el primero siendo el mensaje cifrado y el segundo siendo la firma cifrada. luego igualamos Dcypher a testing.Dcypher.decypherwithsignature, que usará como parámetros Dcypher (donde está el mensaje y firma encriptada) y las claves públicas del receptor.

Código:

```
#include <NTL/ZZ.h>
#include <iostream>
#include "headers/RSA.h"
#include "headers/functions.h"
#include <vector>

using std::cin;
using std::cout;
using std::endl;

int main()
{

    ZZ bits, p, q, e, ea, Na, eb, Nb;
    bool WithSig;
    vector<string>Cypher;
    vector<string>DCypher;
    string alpha, sig, message;

    bits = 1024;
```

```
    alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ,.-(
)abcdefghijklmnopqrstuvwxyz<>*1234567890[]";
    sig = "Aqui va la firma ";
    message =" Aqui va el mensaje";

    char input;
    std::cout<<"RSA";
    std::cout<<"\nPara Test completo (a), para Generar numeros (b), Para cifrar o
descifrar(c)\n";
    std::cin>>input;

    if (input=='a'){
        std::cout<<"\nTEST COMPLETO\n";
        //nuestro abecedario y firma
        std::cout<<"\nEMISOR:\n";
        RSA Emisor(bits,alpha,sig);
        std::cout<<"\nRECEPTOR:\n";
        RSA Receptor(bits,alpha,sig);

        //claves publicas
        ea = Emisor.e;
        Na = Emisor.N;
        eb = Receptor.e;
        Nb = Receptor.N;
        WithSig = true;
        //Emisor cifra
        vector<string>Cypher = Emisor.cypherwithsignature(message, eb, Nb);
        cout << "El mensajeCifrado es: \n" << Cypher.at(0) << "\nfirma:\n" <<
Cypher.at(1) << endl;
        //Receptor decifra
        vector<string>DCypher = Receptor.decypherwithsignature(Cypher, ea, Na);
        cout << "\nMensaje:\n" << DCypher.at(0) << "\nfirma\n" << DCypher.at(1) <<
endl;
    }
//GENERADOR DE NUMEROS

    else if (input=='b'){
        std::cout<<"\nGENERANDO NUMEROS...\n";
        RSA generator(bits,alpha,sig);
    }

//CIFRANDO Y DECIFRANDO

    else if (input=='c'){
```

```
        std::cout<<"\nCifrado o descifrado\n";
        p = conv<ZZ>("clave p predeterminada (para cifrar o descifrar)");
        q = conv<ZZ>("clave q predeterminada");
        e = conv<ZZ>("clave e predeterminada");

        RSA testing(p, q, e, alpha, sig);

        eb=conv<ZZ>("clave publica receptor");
        Nb=conv<ZZ>("N de receptor");
        std::cout<<"\nCLAVES PUBLICAS DE RECEPTOR\ne: "<<eb<<"\nN:
"<<Nb<<"\n";
        std::cout<<"\nPara cifrar(a), para descifrar(b)\n";
        std::cin>>input;
        if (input=='a'){
            //CIFRADO
            std::cout<<"\nCifrando...\n";
            //mensaje
            Cypher=testing.cypherwithsignature(message,eb,Nb);
            std::cout<<"\nMessage: \n"<<Cypher.at(0)<<"\nSignature: \n"<<Cypher.at(1);
        }
        else if (input=='b'){
            //DECIFRADO
            std::cout<<"\nDescifrando...\n";
            //mensaje
            DCypher.push_back("aqui va el mensaje cifrado");
            //firma
            DCypher.push_back("aqui va la firma cifrada");
            DCypher=testing.decypherwithsignature(DCypher,eb,Nb);
            std::cout<<"\nMensaje:\n"<<DCypher.at(0)<<"\nFirma:\n"<<DCypher.at(1);
        }
    }
}
```

2. Generación de claves

Constructores de RSA:
-RSA::RSA(ZZ up, ZZ uq, ZZ ue, string uAlphabet, string uSignature) :

Parámetros:
Recibirá dos variables tipo ZZ up y uq, que serán los dos primos generados de los bits elegidos en el generador, también recibe el string uAlphabet, que será el alfabeto que se

usará al momento de cifrar y descifrar, y por último recibe el string uSignature, que nos servirá como la firma digital de nuestro RSA.

Variables y funcionalidad:

Utilizaremos N de la clase RSA, el cual guardará el valor de p por q, del mismo modo creamos la variable de tipo ZZ fiN cuyo valor está dado por p-1 por q-1, seguidamente usaremos la variable pública d, a la cual le asignaremos la inversa de e por medio de la función inversa, que tomará como parámetros e y fiN, posteriormente se aplicará la función moduloZZ que tomará como parámetros el resultado de la inversa y fiN, esto para evitar que la inversa sea negativa.

Código

```
RSA::RSA(ZZ up, ZZ uq, ZZ ue, string uAlphabet, string uSignature) :
   alphabet(uAlphabet), p(up), q(uq), e(ue), signature(uSignature)
{
   N = p * q;
   ZZ fiN = (p - 1) * (q - 1);
   d = moduloZZ(inversa(e, fiN), fiN);

   std::cout << "p: " << p << std::endl;
   std::cout << "q: " << q << std::endl;
   std::cout << "N: " << N << std::endl;
   //std::cout << "fiN: " << fiN << std::endl;
   std::cout << "e: " << e << std::endl;
   std::cout << "d: " << d << std::endl;
}
```

-RSA::RSA(ZZ nBits, string uAlphabet, string uSignature):

Parámetros:

Un número ZZ que representa de cuantos bits serán nuestras claves, Un string uAlphabet que será el alfabeto del RSA y un string uSignature que será la firma que usaremos en el cifrado del RSA.

Variables y funcionalidad:

La clase Rsa tiene dos atributos llamados alphabet y signature, primero igualamos estos atributos a uAlphabet y uSignature respectivamente. Luego creamos una instancia llamada pseudorandom de la clase **coordinateRNG** usando un constructor que tiene como parámetro nBits.

Para crear p (atributo de la clase RSA) usaremos la función **getprime** que utilizará como parámetro nuestra instancia llamada pseudorandom y un número ZZ que en este caso es p, para crear q (atributo de la clase RSA) usará el mismo procedimiento pero usando q como parámetro de **getprime** en vez de p, al mismo tiempo para hacer el proceso más rápido

usaremos threads en la generación de p y q. N que es un atributo de la clase RSA, será igual a la multiplicación de p y q, fiN será igual a la multiplicación de (p-1) y (q-1).
Para la creación de “e” (atributo del RSA), primero crearemos una instancia llamada blumblum de la clase **blumblumshubRNG** usando un constructor que usará como parámetros a (time(NULL), fiN y nBits). Luego la instancia blumblum usará el método **next()** e igualara el resultado a “e”, haremos un bucle en el que igualamos “e” a **blumblum.next()** y saldrá de este bucle cuando **mcdZZ**(e,fiN) sea igual a 1.
Para la creación de “d” (atributo del RSA), igualamos “d” a la función inversa() que usará como parámetros a (e, fiN) y se le sacará módulo con fiN, “d” será igual a **moduloZZ**(**inversa**(e, fiN), fiN). Finalmente usaremos cout para imprimir p, q, N, e y d.

Código:

```
RSA::RSA(ZZ nBits, string uAlphabet, string uSignature) :
   alphabet(uAlphabet), signature(uSignature)
{
   coordinateRNG pseudorandom(nBits);
   std::thread thread1(getPrime, std::ref(pseudorandom), std::ref(p));
   std::thread thread2(getPrime, std::ref(pseudorandom), std::ref(q));
   thread1.join();
   thread2.join();
   N = p * q;
   ZZ fiN = (p - 1) * (q - 1);
   blumblumshubRNG blumblum(time(NULL), fiN, nBits);
   e = blumblum.next();
   while (mcdZZ(e, fiN) != 1)
   {
      e = blumblum.next();
   }
   d = moduloZZ(inversa(e, fiN), fiN);
   std::cout << "p: " << p << std::endl;
   std::cout << "q: " << q << std::endl;
   std::cout << "N: " << N << std::endl;
   std::cout << "e: " << e << std::endl;
   std::cout << "d: " << d << std::endl;
}
```

o Generación de números aleatorios

Clase blumblumshubRNG, generador de números aleatorios.

ZZ seed, la semilla para el generador.
ZZ m, módulo del generador.
ZZ nBits, cantidad de bits del generador.

ZZ limInf, menor número a ser generado.
ZZ limSup, mayor número a ser generado.

Constructor de la clase: blumblumshubRNG::blumblumshubRNG(time_t initialSeed, ZZ uM, ZZ nBits)

Parámetros

Este constructor recibe 3 parámetros, el tiempo, dos enteros, que son el módulo y la cantidad de bits.

Variables y funcionalidad

Asignamos a “seed”, el valor del casteo de “initialSeed” a long.
Asignamos a “seed”, el valor del **moduloZZ** con m.
Mientras que el maximo comun divisor calculado por **mcdZZ** de “seed” y “m” sea diferente de 1 y la semilla sea mayor a 1,
    Asignamos a “seed”, la suma de “seed” + “seed” + 1, con el **moduloZZ** de “m”.

Código

```
blumblumshubRNG::blumblumshubRNG(time_t initialSeed, ZZ uM, ZZ nBits) :
        m(uM), limInf(rtl(ZZ(2), nBits - 1) - 1), limSup(rtl(ZZ(2), nBits) - 1)
{
        seed = (long)initialSeed;
        seed = moduloZZ(seed, m);
        while (mcdZZ(seed, m) != 1 && seed > 1)
        {
        seed = moduloZZ(seed + seed + 1, m);
        }
}
```

Metodo ZZ blumblumshubRNG::next()

Variables y funcionalidad

Declaramos “temp”, con “seed” por “seed” **moduloZZ** “m”.
Mientras que “temp” sea menor que 1 o, “temp” sea menor que “limInf” o “temp” sea mayor que “limSup”,
    Asignamos a “temp” el resultado del 2 por “temp” + 1 **moduloZZ** de “m”.
Asignamos a “seed”, el nuevo valor de “temp”.

Retorno del método

Se retorna “seed”, el cual es un número completamente nuevo generado aleatoriamente.

Código

```
ZZ blumblumshubRNG::next()
{
```

```
        ZZ temp = moduloZZ(seed * seed, m);
        while (temp < 1 || (temp < limInf || temp > limSup))
        {
        temp = moduloZZ(2 * temp + 1, m);
        }
        seed = temp;
        return seed;
}
```

-Clase coordinateRNG, generador de números aleatorios.

Atributos:

ZZ seed: la semilla del generador.
ZZ nBits: la cantidad de bits del número generado.
ZZ a una: constante, el multiplicador.
ZZ c una: constante, el incremento.
ZZ m una: constante, el módulo.

Constructor de la clase:

-coordinateRNG::coordinateRNG(ZZ bits)

Parámetros:

Recibirá como parámetro una variable de tipo ZZ que representa de cuantos bits será nuestro generador.

Variables y funcionalidad:

Primero igualamos seed a time(NULL) y luego nBits le damos el valor de bits (nuestro parámetro). Luego c lo igualamos a 1, a seed le damos el valor de sí mismo multiplicado por get_coordinate() (un método de nuestra clase), a “a” le damos el valor de **rtl**(3,nBits/2) (rtl es una función de exponenciación) y finalmente a m le damos el valor de **rtl**(2,nBits-1).

Código

```
coordinateRNG::coordinateRNG(ZZ bits) :
   seed(time(NULL)), nBits(bits)
{
   c = ZZ(1);
   seed *= get_coordinate();
   a = rtl(ZZ(3), nBits / 2);
   m = rtl(ZZ(2), nBits - 1);
}
```

Métodos:

-ZZ coordinateRNG::next()

Variables y funcionalidad:

seed será igual a “seed * a + c”, si seed es mayor que m entonces aplicaremos modulo “**moduloZZ**(seed,m)”.

Luego, mientras seed sea menor a (m>>1), seed será igual a “seed<<1” De esta manera haremos que el número generado pseudo aleatoriamente siempre sea de la cantidad de bits que queremos.

Resultado y retorno de la función:
Retorna seed, el cual será un número pseudo aleatorio.

Código
```
ZZ coordinateRNG::next()
{
   seed = seed * a + c;
   if (seed > m)
   {
      seed = moduloZZ(seed, m);
   }
   while (seed < (m >> 1))
   {
      seed = (seed << 1);
   }
   return seed;
}
```

-ZZ coordinateRNG::get_coordinate() Para conseguir las coordenadas del mouse

Variables y funcionalidad:
Declaramos p que será de tipo POINT luego usaremos GetCursorPos y usaremos p como parámetro de tal manera que: **GetCursorPos**(&p).
Declaramos seed de tipo ZZ que será igual a p.x*p.y.

Resultado y retorno de la función:
Retornara seed que es la multiplicación de las coordenadas del mouse.

Código:

```
ZZ get_coordinate()
{
   POINT P;
   GetCursorPos(&P);

   ZZ seed = conv<ZZ>(P.x * P.y);
   return seed;
}
```

o Generación de primos

Función isPrime(ZZ x)

Parámetros
Esta función tiene un parámetro obligatorio, el entero “x” que sera evaluado.

Variables y funcionalidad
Declaramos tres variables auxiliares “res”, “y” e “gdc”.
Declaramos nuestra clase **coordinateRNG**(ZZ(1024)) con el nombre “randro”, sirve para generar numero primos.
Si el **moduloZZ2** de “x” es 0,
    Se retorna *false.*
Asignamos a “y” el valor de un numero aleatorio hecho por el objeto “randro”.**next().**
Asignamos a “gcd” el valor de calcular el maximo comun divisor con la funcion **mcdZZ(x, y).**
Mientras que “gcd” no sea 1,
    Asignamos a “y” el valor de un numero aleatorio hecho por el objeto “randro”.**next().**
    Asignamos a “gcd” el valor de calcular el maximo comun divisor con la funcion **mcdZZ(x, y).**
De esta manera obtenemos un número primo entre sí con “x”.

Resultado y retorno de la función
Aplicamos el teorema de Fermat.
Asignamos a “res” el resultado de elevar, “y”, al exponente “x-1”, con el modulo “x”, gracias a nuestra funcion **powerZZ.**
Si “res” es 1,
    se retorna *true.*
En caso contrario,
    se retorna *false.*

Código
```
bool isPrime(ZZ x)
{
    ZZ res;
    ZZ y;
    ZZ gcd;
    coordinateRNG randnro(ZZ(1024));
    if (moduloZZ2(x) == 0) {
    return false;
    }
    y = randnro.next();
    //y=ZZ(2);
    gcd = mcdZZ(x, y);
    while (gcd != 1) {
    y = randnro.next();
    //y=ZZ(2);
```

```
        gcd = mcdZZ(x, y);
        }
        res = powerZZ(y, (x - 1), x);
        if (res == 1) {
        return true;
        }
        else {
        return false;
        }
}
```

getPrime(coordinateRNG& number, ZZ& param).

Parámetros:

Recibe como parámetros un num de la clase coordinateRNG y un ZZ param, el cual guardará al número generado

Variables y funcionalidad:

Creamos un ZZ temp
Se entra a un do while en el que mientras que la función isPrime que recibe a temp no devuelva un bool true, guardará en temp un número aleatorio mediante .next, para posteriormente entrar en un bucle while en el cual mientras el módulo de 2 del módulo de 10 de temp sea 0, asignará otro número aleatorio también usando .next en number.
Finalmente se iguala param a temp, guardando así el número primo aleatorio generado

Código

```
void getPrime(coordinateRNG& number, ZZ& param)
{
   //int cont = 1;
   ZZ temp;
   do
   {
      temp = number.next();
      while (moduloZZ(moduloZZ(temp, ZZ(10)), ZZ(2)) == 0)
      {
         temp = number.next();
      }
   } while (!isPrime(temp));
   param = temp;
}
```

o Algoritmo de Euclides

ZZ mcdZZ(ZZ x, ZZ y)

Parámetros
Esta función requiere de dos parámetros obligatorios, las variables X y Y.

Variables y Funcionalidad
La función mcd usa el algoritmo de Euclides extendido para poder calcular el máximo común divisor entre dos números (x, y). Todas las variables serán de tipo ZZ, para que sea posible las operaciones entre números grandes

Para esto se hará uso de otras variables auxiliares como como q y r.

q → resultado de la división entera de x/y.
r → residuo de la división de x/y. Se consigue con el uso de nuestra función móduloZZ.

Resultado y retorno de la función
- ➔ Se hará una serie de operaciones hasta que nuestra variable r sea igual a 0.
  - x = y
  - y = r
  - q = x/y
  - r = módulo ZZ(x, y)
  - x = q*y + r
- ➔ Se retornará y, dicha variable almacena el mcd de los números x, y.

Código
```
ZZ mcdZZ(ZZ x, ZZ y)
{
   ZZ q, r;
   q = x / y;
   r = moduloZZ(x, y);

   while (r != 0)
   {
      x = y;
      y = r;
      q = x / y;
      r = moduloZZ(x, y);
      x = q * y + r;
   }
   return y;
}
```

o Inversa

Función inversa(ZZ a, ZZ b), calcular la inversa de dos números

Parámetros
Esta función tiene dos parametros obligatorios, el primero es “a” que es un entero de tipo ZZ, el segundo es “b” que es un entero de tipo ZZ.

Variables, funcionalidad, resultado y retorno de la función
La funcion retorna el segundo elemento del vector resultante de llamar a la funcion **extMcdZZ**, con los parametros (a, b). Este es un entero ZZ que es la inversa de “a” y “b”.

Código

```
ZZ inversa(ZZ a, ZZ b)
{
	return extMcdZZ(a, b).at(1);
}
```

o Euclides extendido

Funcion extMcdZZ(ZZ a, ZZ b) Función para sacar euclides extendido.

Parámetros:
Esta función recibirá dos parámetros y ambos serán de tipo ZZ, el primero será “a” y el segundo será “b”.

Variables y funcionalidad:
Primero declaramos temp que será un vector de tipo ZZ, declaramos q que será la división de a/b y declaramos res que será a-(q*b).
Luego declaramos s1 de tipo ZZ y será igual a 1, declaramos s2 de tipo ZZ y será igual a 0 y declaramos s de tipo ZZ que será igual a s1-(q*s2).
Finalmente declaramos t1 de tipo ZZ que será igual a 0, declaramos t2 de tipo Zz que será igual a 1 y declaramos t que será igual a t1-(q*t2).

Ahora crearemos un bucle “while(res>0)” dentro del bucle haremos lo siguiente:
a será igual a b, b será igual a res, q será igual a “a/b” y res será igual a “a-(q*b)”. Luego s1 será igual a s2, s2 será igual a s y s será igual a s1-(q*s2) y finalmente t1 será igual a t2, t2 será igual a t y t será igual a t1-(q*t2).
Al salir del bucle:
temp hara pushback a: b,s2 y t2.

Resultado y retorno de la función:
Retornara el vector temp, que tendrá 3 variables: b,s2 y t2, si sabemos que mcd(a,b)=d y d=ax+by. entonces b=d, s2=x y t2=y.

Código:

```
std::vector<ZZ> extMcdZZ(ZZ a, ZZ b)
{
    std::vector<ZZ> temp;
    ZZ q = a / b;
    ZZ res = a - (q * b);

    ZZ s1;
    s1 = 1;
    ZZ s2;
    s2 = 0;
    ZZ s = s1 - (q * s2);

    ZZ t1;
    t1 = 0;
    ZZ t2;
    t2 = 1;
    ZZ t = t1 - (q * t2);

    while (res > 0)
    {
        a = b;
        b = res;
        q = a / b;
        res = a - (q * b);

        s1 = s2;
        s2 = s;
        s = s1 - (q * s2);

        t1 = t2;
        t2 = t;
        t = t1 - (q * t2);

    }

    temp.push_back(b);
    temp.push_back(s2);
    temp.push_back(t2);

    return temp;
}
```

3. Formación de Bloques o Llenar ceros

Nuestros procedimientos encargados de l=rellenar los ceros faltantes se encuentran el nuestras funciones de cifrado y descifrado, y lo que hacemos es una comparación de la

longitud del bloque que vamos a ingresar con la longitud que debería tener, cosa que cuando esta longitud sea menor, mediante un for agregaremos esos ceros faltantes.

Para este procedimiento usaremos también funciones de conversión de datos como ***zToString*** y ***StringToVector*** para agregar con la cantidad correspondiente de dígitos cada bloque y respetar las reglas determinadas para el correcto funcionamiento de nuestro proyecto.

o Convertir string a enteros (ZZ)

vector<string> stringToVectorIndex(string message, int ddigits, string alphabet)

Parámetros

Recibe el string message donde se almacena nuestro mensaje, el int ddigits que será el número de dígitos para poder repartir el mensaje en bloques, y el string alphabet qué es el alfabeto que usamos para este proceso de dividir en bloques.

Variables y funcionalidad

Declaramos el vector de strings firstBlock,luego se da ingreso a un for, en el que mientras i no esté al final del vector message, se declara la variable string temp, la cual busca el elemento del mensaje en el alfabeto, y lo añade a temp convertido en string, posteriormente se crea string block el cual se crea con el tamaño de la resta de ddigits y el tamaño de temp, llenando esto de ceros y añadiendo temp al final del mismo, finalmente se añade este string block a first block usando push_back para poder salir del for.

Resultado y retorno de la función

Culminando la función se retorna el vector firstBlock, ya conteniendo nuestro mensaje dividido en bloques

Código

```
vector<string> stringToVectorIndex(string message, int ddigits, string alphabet)
{
    vector<string> firstBlock;
    for (string::iterator i = message.begin(); i != message.end(); i++)
    {
        //IF NOT FOUND WILL RETURN 0
        string temp = std::to_string(alphabet.find(*i));
        //std::cout << temp.length();
        string block = string(ddigits - (int)temp.length(), '0') + temp;
        firstBlock.push_back(block);
    }
    return firstBlock;
}
```

o ZZ a string

zToString (ZZ z)

Parámetros

Será necesaria una variable z, que usaremos al operar el string.

Variables y funcionalidad

Creamos una variable tipo stringstream, la cual llamaremos buffer, y será la que se añade al string junto con z.

Resultado y retorno de la función

Se convierte una variable de tipo ZZ a un string y se retorna dicho string.

Código

```
std::string zToString(ZZ z) {
    std::stringstream buffer;
    buffer << z;
    return buffer.str();
}
```

o Dividir bloques

vector<string> stringToVectorIndex(string message, int ddigits, string alphabet)

Parámetros

Recibe el string message donde se almacena nuestro mensaje, el int ddigits que será el número de dígitos para poder repartir el mensaje en bloques, y el string alphabet qué es el alfabeto que usamos para este proceso de dividir en bloques.

Variables y funcionalidad

Declaramos el vector de strings firstBlock,luego se da ingreso a un for, en el que mientras i no esté al final del vector message, se declara la variable string temp, la cual busca el elemento del mensaje en el alfabeto, y lo añade a temp convertido en string, posteriormente se crea string block el cual se crea con el tamaño de la resta de ddigits y el tamaño de temp, llenando esto de ceros y añadiendo temp al final del mismo, finalmente se añade este string block a first block usando push_back para poder salir del for.

Resultado y retorno de la función

Culminando la función se retorna el vector firstBlock, ya conteniendo nuestro mensaje dividido en bloques

Código

```
vector<string> stringToVectorIndex(string message, int ddigits, string alphabet)
{
    vector<string> firstBlock;
```

```
    for (string::iterator i = message.begin(); i != message.end(); i++)
    {
        //IF NOT FOUND WILL RETURN 0
        string temp = std::to_string(alphabet.find(*i));
        //std::cout << temp.length();
        string block = string(ddigits - (int)temp.length(), '0') + temp;
        firstBlock.push_back(block);
    }
    return firstBlock;
}
```

4. Exponenciación modular

ZZ powerZZ (ZZ b, ZZ e, ZZ m) usando rtl con módulo.

Parámetros:

Se requieren 3 parámetros obligatorios: b, e y m.

Variables y funcionalidad:

Esta función realiza la exponenciación modular con valores ZZ. Siendo:

- ➔ b → la base
- ➔ e → el exponente
- ➔ m → el módulo

Se crea una variable A, la cual será de tipo ZZ, que recibirá un string (“1”) y se le hará un cast a ZZ. Habrá otra variable ZZ llamada S, que será igual a b.

Resultado y retorno de la función:

Se realizará una serie de operaciones hasta que nuestra variable e sea igual a 0.

- ➔ Primero verificamos si el módulo de e es igual a 0, esto lo haremos mediante bitshift, gracias a nuestra función moduloZZ2.
  - ◆ De ser ese el caso, se calculará el módulo de (A*S), en módulo m. Esa será nuestra nueva A
- ➔ Luego, se hará un bitshift hacia la derecha a la variable e.
- ➔ Y realizamos una última verificación, si e no es igual a 0.
  - ◆ De ser ese el caso, calculamos el módulo de (S*S) en módulo m. Siendo ese resultado nuestra nueva S.
- ➔ Finalmente retornamos A, que es el resultado de nuestra exponenciación modular.

Código

```
ZZ powerZZ(ZZ b, ZZ e, ZZ m)
{
    ZZ A = conv<ZZ>("1");
    ZZ S = b;

    while (e != 0)
    {
```

```
        if (moduloZZ2(e) == 1)
        {
            A = moduloZZ(A * S, m);
        }
        e = e >> 1;
        if (e != 0) S = moduloZZ((S * S), m);
    }
    return A;
}
```

5. Función de cifrado

Función RSA::cypher(string message, ZZ publicKey, ZZ Nb, bool withSig)

Parámetros

Recibe tres parámetros obligatorios y uno opcional, el primero es el mensaje en forma de string, el segundo es la clave pública en ZZ, el N de las claves generadas por el receptor externo también en ZZ y el parámetro opcional es un booleano que indica si se tiene que hacer un cifrado normal o con rúbrica y firma.

Variables y funcionalidad

Declaramos el entero “availableDigits”, y le asignamos el valor de la longitud de la longitud del alfabeto esto nos sirve para conocer cuán grandes deben de ser los primeros bloques.
Declaramos el entero “nDigits”, utilizamos la función **zToString** que convierte un número entero ZZ a un string, y le sacamos la longitud a este string para así saber cuantos digitos tiene N.
Crearemos un bucle que dará vueltas mientras **modulo(**aviableDigits*message.length(), nDigits-1**)** no sea igual a 0 y hará message.append(“ “) en cada iteración.
Declaramos el vector de strings “temp”, le asignamos el valor que resulta de dividir el mensaje en bloques delimitados por availableDigits en este proceso se busca cada carácter en el alfabeto, se coloca en un bloque y se aumentan ceros para así los bloques tengan un tamaño ideal, luego este vector es convertido en un string, que luego será dividido en bloques delimitados por nDigits - 1.
Declaramos el vector de strings “tempCypher”, donde almacenaremos los bloques cifrados.

Iteramos desde el comienzo del vector temp hasta su final, en el iterador “i”. En cada iteración,

Declaramos “temp1”, que convierte el iterador a un string.

Declaramos “zTemp”, que convierte un c_string a ZZ, pero antes cambiamos temp1 a c string para que no exista problema alguno.

Declaramos “m”, variable en la cual almacenaremos las exponenciaciones.

Si withSig es falso,

Asignamos a “m”, el resultado de **powerZZ**, que es nuestra función de exponenciación modular, elevamos “m” a la clave pública del receptor con módulo N del receptor.

Asignamos a “temp1”, el valor resultante de convertir “m” a un string.

Hacemos push_back al vector “tempCypher”, con el resultado de agregar ceros a “temp1” para que tenga la cantidad necesaria para el bloque de acuerdo con la longitud de “N” del receptor.

En caso contrario,

Asignamos a “m”, el resultado de **powerZZ** elevamos “m” a nuestra clave privada con módulo de nuestra N. Asi obtenemos la rubrica del bloque.

Asignamos a “m”, el resultado de **powerZZ**, que es nuestra función de exponenciación modular, elevamos “m” a la clave pública del receptor con módulo N del receptor.

Asignamos a “temp1”, el valor resultante de convertir “m” a un string.

Hacemos push_back al vector “tempCypher”, con el resultado de agregar ceros a “temp1” para que tenga la cantidad necesaria para el bloque de acuerdo con la longitud de “N” del receptor.

Resultado y retorno de la función

La función retorna el resultado de convertir el vector “tempCypher” a un string, lo cual sería el mensaje ya cifrado.

Código

```
string RSA::cypher(string message, ZZ publicKey, ZZ Nb, bool withSig)
{
	int availableDigits = std::to_string((int)alphabet.length()).length();
	int nDigits = (int)zToString(Nb).length();
	vector<string> temp = stringToVector(vectorToString(stringToVectorIndex(message, availableDigits, alphabet)), nDigits - 1);
	vector<string> tempCypher;
	for (vector<string>::iterator i = temp.begin(); i != temp.end(); i++)
	{
	string temp1 = *i;
	ZZ zTemp(INIT_VAL, temp1.c_str()); ZZ m;
	if (!withSig){
	m = powerZZ(zTemp, publicKey, Nb);
	temp1 = zToString(m);
	tempCypher.push_back(string((nDigits) - (int)temp1.length(), '0') + temp1);
	}
	else {
	//std::cout << "\nCon firma\n";
	m = powerZZ(zTemp, this->d, this->N);
	m = powerZZ(m, publicKey, Nb);
	temp1 = zToString(m);
	tempCypher.push_back(string((int)zToString(Nb).length()-(int)temp1.length(), '0')+temp1);

	}
	//std::cout << string(nDigits - (int)temp1.length(), '0') + temp1 << std::endl;
	//tempCypher.push_back(string(nDigits - (int)temp1.length(), '0') + temp1);
	return vectorToString(tempCypher);
```

}

6. Función de descifrado

String RSA::decypher (string message, ZZ publickey, ZZ Na, bool withSig)

Parámetros:

La función recibe como parámetros un string que sería el mensaje a descifrar, una clave pública de a, se podría decir que del emisor y un booleano para saber si el mensaje contiene una firma o no.

Variables y funcionalidad:

En cuanto a variables, tendremos un string llamado toReturn, que será el string que devolverá nuestra función, una variable availableDigits que tomará los dígitos del tamaño del alfabeto llamando a la función ***to_string***. Tendremos otra variable del tipo entera nDigits que tomará la cantidad de dígitos de N declarado en el constructor. Además tendremos una variable tipo string temp 1, una variable zTemp de tipo ZZ, una variable tipo ZZ llamada m y un string auxiliar. Finalmente contamos con dos variables vector<string>, a una de ellas le asignamos el vector creado a partir de la función ***stringToVector*** que recibirá el string mensaje y el entero creado anteriormente nDigits y la otra sola la declararemos.

Seguido a esto entraremos a un for que iterará en temp con una variable i hasta que esta misma llegué al fin de temp. A continuación usaremos nuestro booleano para comprobar que sea una firma o un mensaje y dependiendo de esto elevaremos nuestro m con la función ***powerZZ*** recibiendo los parámetros adecuados según sea el caso del booleano. Si el booleano es falso entonces elevará m a nuestra clave privada y nuestro N, si el booleano es verdadero elevará m a nuestra clave privada y nuestro N y luego elevará m a la clave pública del receptor con módulo N del receptor.

Guardaremos esta potenciación en temp1 ayudándonos con la función ***zToString*** que convertirá m en un string y declaramos un string llamado auxiliar, luego iremos aumentando los ceros necesarios según sea el caso según nuestras variables y sus valores. Para esto declaramos la variable int tempDigits y usaremos nuestro booleano WithSig nuevamente. Si WithSig es verdadero entonces tempDigits será igual a **Ztostring**(Na).length()-1 qué es la cantidad de dígitos - 1 del N del receptor. Si es el caso contrario entonces tempdigits será igual a nDigits-1, qué es la cantidad de dígitos de nuestro N - 1. Finalmente, usaremos tempdigits para agregar 0's según sea necesario e igualamos el resultado a auxiliar.

Con esto desarrollado, a tempDecypher le hacemos un push back recibiendo esta misma instrucción como parámetro a auxiliar para luego guardar este valor dentro de temp con la función ***stringToVector*** y con esta variable haremos un for con la misma idea del for anterior, es decir con una variable i recorriendolo para que con una comprobación de los dígitos necesarios, vayamos agregando valores al string toReturn para terminar retornándolo

Resultado y retorno de la función

Retornaremos el valor toReturn que sería nuestro mensaje descifrado satisfactoriamente.

```
string RSA::decypher(string message, ZZ publickey, ZZ Na, bool withSig)
{
    string toReturn;
    int availableDigits = std::to_string((int)alphabet.length()).length();
    int nDigits = (int)zToString(N).length();
    vector<string> temp = stringToVector(message, nDigits);
    vector<string> tempDecypher;
    for (vector<string>::iterator i = temp.begin(); i != temp.end(); i++)
    {
        string temp1 = *i;
        ZZ zTemp(INIT_VAL, temp1.c_str());
        ZZ m;
        if (!withSig){
            m = powerZZ(zTemp, this->d, this->N);
        }
        else {
            m = powerZZ(zTemp, this->d, this->N);
            m = powerZZ(m, publickey, Na);
        }
        // temp1 = zToString(powerZZ(zTemp, d, N));ón
        temp1 = zToString(m);
        string auxiliar;
        if (withSig){
            if((int)zToString(Na).length()-1!=temp1.length()){
```

```
            auxiliar=string("0")+temp1;
        }
        else{
            auxiliar=temp1;
        }
        //auxiliar=string(((int)zToString(Na).length()-1)-(int)temp1.length(), '0')+temp1;
        //auxiliar=temp1;
    }
    else{
        auxiliar = string((nDigits-1) - (int)temp1.length(), '0') + temp1;
    }
    //string auxiliar=temp1;
    while (modulo(auxiliar.length(), availableDigits) && i == temp.end() - 1)
    {
        auxiliar.erase(0, 1);
    }
    tempDecypher.push_back(auxiliar);

}
temp = stringToVector(vectorToString(tempDecypher), availableDigits);
for (vector<string>::iterator i = temp.begin(); i != temp.end(); i++)
{
    string temp1 = *i;
    int iTemp = std::stoi(temp1);
    if (iTemp < (int)alphabet.length()) toReturn.append(alphabet, iTemp, 1);
}
```

```
    return toReturn;
}
```

7. Firma digital

Función RSA::cypherwithsignature(string message, ZZ publicKey, ZZ Nb)

Parámetros

Se pasan como parámetros el mensaje que se cifrará, nuestra clave pública en ZZ y el N de las claves generadas por el receptor externo también en ZZ.

Variables y Funcionalidad

Se creará un vector de tipo string, que llamaremos temp, dicho vector almacenará nuestro mensaje cifrado por la función cypher, y también se añade la firma cifrada por la misma función cypher.

Resultado y retorno de la función

Tras el cifrado de ambas partes y el añadido de estas al vector, se retornará el mismo.

Código

```
vector<string> RSA::cypherwithsignature(string message, ZZ publicKey, ZZ Nb) {
    vector<string>temp; temp.push_back(cypher(message, publicKey, Nb));
    temp.push_back(cypher(signature, publicKey, Nb, true));
    return temp;
}
```

RSA::decypherwithsignature(vector<string>message, ZZ publickey, ZZ Na)

Parámetros

Los parámetros de la funcion son un vector tipo string message que guarda el mensaje encriptado, una variable tipo ZZ publickey que tendrá almacenada la clave pública y la variable tipo ZZ Na

Variables y funcionalidad

Esta función nos devolverá el vector tipo string temp, el cual crearemos al inicio de la misma, al tener temp se le añade el mensaje descifrado mediante un push_back, el cual toma la función decypher que tomará por parámetros el mensaje, la clave pública y Na.

Posteriormente mediante un push_back se añadirá la firma a temp, con la misma función decypher, pero agregando como parámetro el booleano true para señalar que esta será la firma.

finalmente retornará temp, ya habiendo recibido mensaje y firma.

Código

```
vector<string> RSA::decypherwithsignature(vector<string>message, ZZ publickey, ZZ Na) {
    vector<string>temp; temp.push_back(decypher(message.at(0), publickey, Na));
    std::cout<<"\n mensaje: \n"<<temp.at(0)<<"\n";
    temp.push_back(decypher(message.at(1), publickey, Na, true));
    return temp;
}
```

8. Funciones extra

Modulo (int a, int b)

Parámetros

Dos enteros, del que sacaremos el módulo, es decir a módulo b.

Variables y funcionalidad

Usaremos una variable entera r que asumirá el resto entre a y b y luego de esto si r es menor que 0, arreglaremos ese valor para que r siempre sea positivo sumándole a esa misma r un 1 o 0 dependiendo de la condición explicada anteriormente multiplicada por el mismo b.

Resultado y retorno de la función:

Devolveremos el resto positivo de estos dos números.

```
int modulo(int a, int b)

{

    int r = a - b * (a / b);

    r = r + (r < 0) * b;

    return r;

}
```

ZZ moduloZZ(ZZ a, ZZ b)

Parámetros
Recibe como parámetros dos variables tipo ZZ a y b, el primero será el número al que se le sacará el módulo y el segundo será el módulo al que se someterá a.

Variables y funcionalidad
Creamos ZZ q, el cual será la división de a entre b, también creamos ZZ res que se le asigna a menos q por b, si este resultado es menor a cero se restará 1 a q y se repetirá la misma operación en res, hasta que se deje de cumplir la condición, Finalmente se retorna res que será a en módulo de b.

Código
```
ZZ moduloZZ(ZZ a, ZZ b)
{
   ZZ q = a / b;
   ZZ res = a - (q * b);
   if (res < 0)
   {
      q--;
      res = a - (q * b);
   }
   return res;
}
```

Función moduloZZ2(ZZ a) Versión optimizada del módulo

Parámetros:
Tendrá como parámetro obligatorio, un número de tipo ZZ.

Variables y funcionalidad:
declaramos la variable r de tipo ZZ que será igual a “a-((a>>1)<<1)”, luego r será igual a “r+((r<0)<<1)”

Resultado y retorno de la función:
Retornaremos r, que será nuestra variable después de haberle sacado módulo.

Código:
```
ZZ moduloZZ2(ZZ a)
{
   ZZ r = a - ((a >> 1) << 1);
   r = r + ((r < 0) << 1);
   return r;
```

}

Función vectorToString(vector<string> blocks) Convierte vector a string

Parámetros:
Nuestro único parámetro es un vector de tipo string, que será uno de los bloques que creemos en nuestros métodos de la clase RSA

Variables y funcionalidad:
Declaramos a final que será de tipo string.
Si !blocks.empty() (si no está vacío), entonces recorreremos todo el vector mediante un for(vector<string>::iterator i = blocks.begin(); i != blocks.end(); i++) y en cada iteración haremos lo siguiente: final.append(*i) . Por lo que, al final de este bucle tendremos todos los elementos de blocks dentro de nuestra nueva variable final.

Resultado y retorno de la función:
Retornaremos final, que será un string que contiene todos los elementos del vector blocks.

Código:
```
string vectorToString(vector<string> blocks)
{
   string final;
   if (!blocks.empty())
   {
      for (vector<string>::iterator i = blocks.begin(); i != blocks.end(); i++)
      {
         final.append(*i);
      }
   }
   return final;
}
```

StringToVector(string message, int ddigits)

Parámetros

Recibiremos un string que será el mensaje que convertiremos a vector y un entero ddigits que será la cantidad que poco a poco iremos añadiendo.

Variables y funcionalidad

Usaremos una variable del tipo vector<string> llamada blocks que en un for usando una variable entera int iremos recorriendo el mensaje y añadiendolo en el vector anteriormente creado respetando el valor de ddigits.

Resultado y retorno de la función:

La función devolverá la variable previamente creada, llamada blocks en la que poco a poco agregamos las variables necesarias que pedimos en los parámetros.

```
vector<string> stringToVector(string message, int ddigits)
{
   vector<string> blocks;
   for (int i = 0; i < (int)message.length(); i += ddigits)
   {
      blocks.push_back(string(message, i, ddigits));
   }
   return blocks;
}
```

Función rtl(ZZ b ,ZZ e), calcular potencia sin modulo

Parámetros

Esta función tiene dos parámetros obligatorios, el primero es “b” que es un entero de tipo ZZ que es la base, el segundo es “e” que es un entero de tipo ZZ que es el exponente.

Variables y funcionalidad

Declaramos dos variables auxiliares “A” con el valor entero 1 y “S” con el valor de “b”.

Mientras que “e” no sea 0,
- Si el **moduloZZ2** de “e” es 1,
  - Asignamos a “A”, el resultado de multiplicar “S” por “A”.
- Asignamos a “e”, el resultado de hacer un corrimiento de bits a la derecha de “e”.
- Si “e” no es 0,
  - Asignamos a “S”, el resultado de multiplicar “S” por “S”.

Resultado y retorno de la función

Retornamos A, que es el resultado de elevar “b” al exponente “e”.

Código

```
ZZ rtl(ZZ b, ZZ e)
{
        ZZ A = conv<ZZ>("1");
        ZZ S = b;

        while (e != 0)
        {
        if (moduloZZ2(e) == 1)
        {
        A = A * S;
        }
```

```
        e = e >> 1;
        if (e != 0) S = S * S;
        }
        return A;
}
```

9. Prueba