Universidad Católica
San Pablo

CIENCIA DE LA COMPUTACIÓN

IMPLEMENTACIÓN DEL ALGORITMO RSA

ÁLGEBRA ABSTRACTA

DANIELA CHAVEZ AGUILAR

GIULIA NAVAL FERNANDEZ

PAOLO DELGADO VIDAL

RODRIGO TORRES SOTOMAYOR

PABLO CARAZAS BARRIOS

AÑO 2021

RESUMEN

En este trabajo se hace una exposición de la estructuración del RSA que el grupo realizó. Asimismo, se brinda una explicación de las funciones utilizadas en distintas partes del algoritmo, donde se habla de su funcionamiento y cómo fueron implementadas.

1. INTRODUCCIÓN

En el presente trabajo se hará una exposición y explicación de nuestra implementación del algoritmo RSA. En dicha explicación se mostrará: la estructura del main; la generación de las claves que se subdivide en: generación de aleatorios, generación de primos, algoritmo de Euclides y la inversa; formación de bloques que se subdivide en: la completación de ceros, la conversión de string a ZZ, de ZZ a string y la división de bloques; exponenciación modular; función cifrado y descifrado y, finalmente, la firma digital.

2. CONTENIDO TEÓRICO

2.1 ESTRUCTURA DEL MAIN

- Primero, generamos 2 objetos que serán el emisor y receptor, se les pasa el número de bits y estos generarán sus respectivas claves. Luego de esto, intercambian sus claves públicas (e y n) y las guardan en el atributo creado para recibir las claves del otro objeto. Una vez hecho esto, Bob (nuestro emisor) cifra el mensaje para que Alice (receptor) lo descifre. Para cifrar y descifrar a personas en otra pc, se crean además 2 variables e y n para recibir sus claves y luego pasarlas al objeto.

Código

```
int main()

{ ZZ e,n;

  RSA Alice(512);

  RSA Bob(512);

  Bob.eB=Alice.e;

  Bob.nB=Alice.n;

  Alice.eB=Bob.e;

  Alice.nB=Bob.n;

string m="Obi Wan Kenobi dice Tu eras el elegido El que destruiria a los Sith no el que se uniria a ellos Se suponia que ibas a traer el equilibrio a la Fuerza no a hundirla en la oscuridad Darth Vader Te odio Obi Wan Kenobi Eras mi hermano Anakin Yo te queria Fue ahi donde comenzo la triste historia entre Obi Wan Kenobi y Darth Vader Todos los espectadores sueltan una lagrimita";

  string m_C=Bob.cifrado(m);

  cout<<m_C<<endl;

string m_D=Alice.descifrado(m_C);

  cout<<m_D<<endl;


}
```

2.2 GENERACIÓN DE CLAVES

1. Generación de números aleatorios.

- El algoritmo utilizado para la generación de números aleatorios fue el Blum-Blum-Shub, ya que fue el que mejor desempeño tuvo en las pruebas realizadas anteriormente y por ser criptográficamente seguro, aunque dentro de la amplia gama de generadores de números aleatorios, no sea muy rápido. Este algoritmo requiere de la generación de una semilla y 2 “primos” de Blum.

- Para la generación de la semilla, se utilizó la posición del cursor, cuyas coordenadas fueron multiplicadas, este valor se adecuó posteriormente al número de bits requerido, haciendo que esté en el rango entre 2^{bits-1} y $2^{bits} - 1$.

```
ZZ seed_generator(ZZ bits)
{
      POINT P;
      GetCursorPos(&P);
      ZZ seed;
      seed = P.x * P.y +1;
      seed =exp_bitsminus1 + divi(seed,intervalo);
      return seed;
}
```

- Por otro lado, para la generación de los primos de Blum, primero se obtuvieron otros números aleatorios, con otra semilla generada a partir del reloj del sistema (nanosegundos) y un generador de aleatorios con un nivel menor de complejidad, como es el Splitmix, que se basa en “recetas numéricas” y la operación xor.

```
uint64_t nanoseconds(){
```

```
        uint64_t seconds;

        auto nanoseconds =
        std::chrono::duration_cast<std::chrono::nanosecond
        s>(std::chrono::system_clock::now().time_since_epo
        ch()).count();

        return nanoseconds;

}

static uint64_t t_seed = nanoseconds();

uint64_t splitmix() {

    uint64_t z = (t_seed += 0x9e3779b97f4a7c15);

    z = (z ^ (z >> 30)) * 0xbf58476d1ce4e5b9;

    z = (z ^ (z >> 27)) * 0x94d049bb133111eb;

    return z ^ (z >> 31);

}

ZZ random(ZZ bits){

        ZZ randomn = ZZ(splitmix());

        randomn = exp_bitsminus1 +
        divi(randomn,intervalo);

        return abs(randomn);

}
```

- Este número se pasa al generador de primos Blum, en el cual, se generará un nuevo aleatorio mientras el número no sea congruente a 3 en módulo 4.

```
ZZ random_blum_prime(ZZ bits){

        ZZ newnum;

        newnum = random(expbits);

        while(newnum <
        exp_bitsminus1||divi(newnum,conv<ZZ>(4))!=3){

        newnum = random(expbits);

        }

        return newnum;

}
```

- Generada la semilla y los primos, se obtendrá los aleatorios para el RSA con el algoritmo Blum-Blum-Shub, que obtendrá una secuencia binaria a partir del bit menos significativo de una serie de potencias al cuadrado de la semilla en módulo n, siendo n el producto de los 2 primos generados.

```
ZZ random_numberBBS(ZZ bits){
      ZZ p,q,n,seed,nextRandNum;
      string resultado="";
      p = random_blum_prime(bits);
      q = random_blum_prime(bits);
      seed = seed_generator(bits);
      n = p*q;
      resultado += "1";
      while(bits != 1){
      nextRandNum = divi((seed*seed),n);
      seed = nextRandNum;
      resultado += Num2Str(par(nextRandNum));
      bits--;
      }
      return binary2ZZ(resultado);}
```

- Finalmente, esta secuencia de bits se retorna convertida a su valor numérico mediante la función binary2ZZ, que se muestra a continuación:

```
ZZ binary2ZZ(string binario){
      ZZ decimal, multiplicador = conv<ZZ>(1);
      for(int i=0;i<bitz;i++){
      char dig = binario[i];
      if(dig=='1'){
            decimal+=multiplicador;
      }
      multiplicador<<=1;
      }
```

```
    return decimal;
}
```

2. Generación de primos.

- Para la generación de primos se utilizó el algoritmo de Miller-Rabin por ser el más eficiente dentro de las pruebas.

```
bool miller_rabin(ZZ n,long t){
    if (n<2) return false;
    if (n==2 || n==3) return true;
    ZZ a,y,i,j,s,r;
    s=0;
    r=n-1;
    while(r>1 && par(r)){
        r>>=1;
        s++;
    }
   for(i=1;i<=t;i++){
        a=ZZ(2)+divi(seed_generator(bitz), n-3);
        y=right2left_binary_modexp(a,r,n);
        if(y!=1 && y!=n-1)
        {
            j=1;
            while(j<=s-1 && y!=n-1)
            {
                y=divi(y*y,n);
                if(y==1)return false;
                j++;
            }
            if(y!=n-1)return false;
     }
    }
```

```
    return true;
}
```

❖ Esta función se llama en la generación de primos, para verificar la primalidad de un número generado con el algoritmo Blum-Blum-Shub. De no ser primo, se vuelve a llamar al generador de aleatorios hasta encontrar un número que lo sea.

```
ZZ GenPrimo(long bits){
    ZZ p;
    p=random_numberBBS(bitz);
    bool es_primo=miller_rabin(p,2);
    while(!es_primo){
        p=random_numberBBS(bitz);
        es_primo=miller_rabin(p,2);
    }
return p;
}
```

3. Algoritmo de Euclides.

❖ Para poder hallar el máximo común divisor dentro de nuestro rsa, con el fin de buscar números que tengan inversa, haremos uso del algoritmo de mcd que consideramos más eficiente, el algoritmo de euclides binario. Para ello, le pasaremos por parámetro dos números, el candidato a primer número de la clave pública “e” y el número phi de n “oN”, y posteriormente pasará a reducir ambos números mediante desplazamientos binarios a la derecha, según las siguientes tres reglas:

- Si a y b son pares: $2mcd(\frac{a}{2}, \frac{b}{2})$
- Si a es par y b es impar: $mcd(\frac{a}{2}, b)$
- Si a y b son impares: $mcd(\frac{|a-b|}{2}, b)$ = $mcd(\frac{|a-b|}{2}, a)$

Algoritmo de Euclides Binario

Código C++:

```
ZZ BinaryGCD(ZZ x, ZZ y){
```

```
 ZZ g=conv<ZZ>(1);
 while(par(x) && par(y)){
x>>=1;y>>=1;g<<=1;
 }
 while(x!=0){
while(par(x)){
      x>>=1;
}
while(par(y)){
      y>>=1;
}
ZZ t=abs(x-y); t>>=1;
if (x>=y){
      x=t;
}
else{
      y=t;
}
 }
 return g*y;
}
```

4. **<u>Inversa - Algoritmo de Euclides extendido.</u>**

- Para poder hallar la inversa que se utilizará para la clave privada, decidimos emplear el algoritmo de euclides binario, ya que según las pruebas realizadas previamente, lo consideramos como el más eficiente. Para ello, pasaremos por parámetro dos números, el número “e” de la clave pública y el número n, y procederá a operar con ambos números, acortando significativamente el proceso gracias al desplazamiento binario, y finalmente retornando tanto el mcd como los valores “x” y “y” del teorema, de las cuales solo emplearemos “x” como la inversa.

  Algoritmo Extendido de Euclides Binario

  Código C++:

```
vector<ZZ> ExtendedBinaryGCD(ZZ x, ZZ y){
  ZZ g=conv<ZZ>(1);
  while(par(x) && par(y)){
x>>=1;y>>=1;g<<=1;
  }
```

```
  ZZ
u=x,v=y,A=conv<ZZ>(1),B=conv<ZZ>(0),C=conv<ZZ>(0),D=conv
<ZZ>(1);
  while(u!=0){
while(par(u)){
      u>>=1;
      if (par(A) && par(B)){
      A>>=1;B>>=1;
      }
      else{
      A+=y;A>>=1;
      B-=x;B>>=1;
      }
}
while(par(v)){
      v>>=1;
      if (par(C) && par(D)){
      C>>=1;D>>=1;
      }
      else{
      C+=y;C>>=1;
      D-=x;D>>=1;
      }
}
if (u>=v){
      u-=v;A-=C;B-=D;
}
else{
      v-=u;C-=A;D-=B;
}
  }
  vector<ZZ> result{g*v,C,D};
  return result;
}
```

2.3 FORMACIÓN DE BLOQUES

1. **Llenar 0's.**

```
string Completar0s(ZZ n, int digs){
    string num = Num2Str(n);
    int n_ceros = digs-num.size();
    if (3>n_ceros&&n_ceros>0){
```

```
    string ceros(n_ceros,'0');
    num = ceros+num;}
    return num;
}
```

2. **Conversión de string a ZZ.**

- Se utilizó una variable del tipo *istringstream* que simplifica el paso de una variable del tipo cadena a una numérica.

```
ZZ Str2Num(string a){
    istringstream num(a);
    ZZ n;
    num>>n;
    return n;
}
```

3. **Conversión de ZZ a string.**

- Se utilizó una variable del tipo *ostringstream* a la que se le pasa el entero(ZZ) y se retorna la conversión a cadena del stream.

```
string Num2Str(ZZ a){
    ostringstream str;
    str<<a;
    return str.str();
}
```

4. **Dividir bloques.**

- Para dividir el bloques el mensaje y la firma, tanto en el cifrado como en el descifrado, el proceso es similar: se crea una variable k del tipo long con el tamaño que debe tener el bloque, luego en un for, se crea subcadenas del mensaje, desde i tomando k caracteres, este bloque es el que se cifrará o descifrará con exponenciación modular y las claves correspondientes para después completarse con 0's y agregarse a la cadena en que se que guardará el mensaje que se retornará. En cada iteración i avanza en k unidades.

```
string cifrado(string m)
{   ostringstream c,r;
long num_dig=Num2Str(alfabeto.size()-1).size();
long k=Num2Str(nB).size()-1;
////cifrado mensaje////
while(divi((m.size()*num_dig),k)!=0)
```

```
        m+=' ';
    for (long i=0; i<m.size(); i++){
    long l=alfabeto.find(m[i]);
    c<<Completar0s(l,num_dig);
    }
    string C=c.str();
    string Cifrado="";
    for (long i=0; i<C.size(); i=i+k){
        string bloque=C.substr(i,k);
        ZZ cypher = right2left_binary_modexp (Str2Num
        (bloque), eB, nB);
        Cifrado+= Completar0s(cypher,k+1);
    }
    ...
```

2.4 EXPONENCIACIÓN MODULAR

- **Exponenciación modular binaria right to left**

  Este algoritmo se fundamenta en el hecho de que si el exponente e es par, se puede calcular ge mod n como (ge/2 * ge/2) mod n, reduciendo el número de multiplicaciones necesarias a 2t, siendo t el número de bits que se requieren al convertir e a su representación binaria. En este caso, la representación binaria se obtiene poco a poco mediante divisiones entre 2 (shifts) y verificaciones de paridad.

  Código C++:

```
ZZ right2left_binary_modexp(ZZ g, ZZ e, ZZ n){
  ZZ A=conv<ZZ>(1), S=g;
  while (e!=0){
if (!par(e))
      A=divi(A*S,n);
e>>=1;
if (e!=0)
      S=divi(S*S,n);
  }
  return A;
}
```

2.5 FUNCIÓN CIFRADO

- En la función cifrado se recibe el mensaje, el cual primero se convierte en una cadena con las posiciones en el alfabeto de cada carácter. Posteriormente, en un for, se dividirá esta cadena en bloques del tamaño de la clave n del receptor (nB) menos 1. Cada bloque se cifrará elevándolo a la clave e del receptor (eB), módulo nB. Finalmente, se completará el bloque con 0 's para que sea del tamaño de nB y se añadirá a la cadena que contendrá el mensaje cifrado. Dentro de esta función, se añadirá la firma digital luego de cifrar el mensaje.

```
string cifrado(string m)
{   ostringstream c,r;
     long num_dig=Num2Str(alfabeto.size()-1).size();
     long k=Num2Str(nB).size()-1;
     ////cifrado mensaje////
     while(divi((m.size()*num_dig),k)!=0)
          m+=' ';
     for (long i=0; i<m.size(); i++){
     long l=alfabeto.find(m[i]);
     c<<Completar0s(l,num_dig);
     }
     string C=c.str();
     string Cifrado="";
     for (long i=0; i<C.size(); i=i+k){
          string bloque=C.substr(i,k);
          ZZ cypher = right2left_binary_modexp (Str2Num
          (bloque),eB,nB);
          Cifrado+= Completar0s(cypher,k+1);
     }
     /////rúbrica y firma/////
     ...
     return Cifrado+" "+Firma;
}
```

2.6 FUNCIÓN DESCIFRADO

- En la función descifrado se recibe el mensaje cifrado, el cual se dividirá en bloques del tamaño de la clave n (propia). Cada bloque se descifrará elevándolo a la clave d, módulo n. Luego, se completará el

bloque con 0 's para que sea del tamaño de n-1 y se añadirá a la cadena que contendrá el mensaje descifrado. Finalmente se tomarán bloques del tamaño de la cantidad de caracteres del alfabeto y se traducirá cada bloque al carácter que se encuentre en la posición que indica el bloque dentro del alfabeto. Dentro de esta función, también se descifra la firma digital luego de descifrar el mensaje.

```
string descifrado(string m)
      {   long p = m.find(" ");
      string msje= m.substr(0,p);
      string firma=m.substr(p+1,m.size()-1);
      /////Descifrado//////
      string Descifrado="",D="";
      long long num_dig = Num2Str (alfabeto.size() - 1)
      .size();
      long long k=Num2Str(n).size();
      for (long long i=0; i<msje.size(); i=i+k){
            string bloque=msje.substr(i,k);
            ZZ decode=Expmod_rchino(Str2Num(bloque),d,n);
            D+= Completar0s(decode,k-1);
      }
      for (long long i=0; i<D.size(); i=i+num_dig){
            string bloque=D.substr(i,num_dig);
            long long pos=Str2Int(bloque);
            char a=alfabeto[pos];
            Descifrado+=a;
      }
      ////Firma y rúbrica/////
      ...
      return Descifrado+" "+Rubrica;
      }
```

2.7 FIRMA DIGITAL

Parte del cifrado

- Dentro de la función de cifrado se añade una firma que contenga al menos tantos caracteres como 10 veces la cantidad de bits con la que se está trabajando, a esta cadena se le agregan espacios hasta que se pueda dividir en bloques de n-1 caracteres en el primer cifrado y n en el segundo. Esta cadena se divide en bloques de n-1 y cada bloque se cifra elevándolo a la clave d en módulo n, luego se completa con 0s para completar n caracteres y añadir el bloque a la rúbrica. La rúbrica se vuelve a cifrar, dividiéndola en bloques de nB-1 caracteres, elevándo cada bloque a eB en módulo nB, completando con 0s hasta completar nB caracteres y sumando cada uno a la cadena con la firma

cifrada. Al final se retorna el mensaje cifrado y la firma cifrados en una cadena, separados por un espacio.

```
...
/////rúbrica y firma/////
      long k2=Num2Str(n).size()-1;
      string d_F=Completar_datos(datos,k2);
      while(divi((d_F.size()*num_dig),k2)!=0 ||
divi(((k2+1)*((d_F.size()*num_dig)/k2)),k)!=0){
           d_F+=' ';}
      for (long i=0; i<d_F.size(); i++){
           long l=alfabeto.find(d_F[i]);
           r<<Completar0s(l,num_dig);
      }
      string R=r.str();
      string Rubrica="";
      for (long i=0; i<R.size(); i=i+k2){
           string bloque=R.substr(i,k2);
           ZZ rubric=Expmod_rchino(Str2Num(bloque),d,n);
           Rubrica+= Completar0s(rubric,k2+1);
      }
      string Firma="";
      for (long i=0; i<Rubrica.size(); i=i+k){
           string bloque=Rubrica.substr(i,k);
           ZZ
signature=right2left_binary_modexp(Str2Num(bloque),eB,nB
);
           Firma+= Completar0s(signature,k+1);
      }
      return Cifrado+" "+Firma;
```

Parte del descifrado

- Dentro de la función de descifrado se obtiene también la rúbrica descifrada. Primero se descifra la firma, dividiéndola en bloques de tamaño n, aplicando exponenciación modular con las claves d y n, y completando con 0s hasta completar n-1 caracteres para agregarlo a la cadena con la firma descifrada. Luego, esta firma se vuelve a dividir en bloques, esta vez de tamaño nB(n del emisor), se descifra aplicando exponenciación modular con las claves e y n del emisor y se completa con 0s hasta completar nB-1 caracteres y sumar a la cadena con la rúbrica descifrada. Finalmente, se traduce la cadena,

separándola en bloques del tamaño del mayor índice dentro del alfabeto, y tomando cada bloque como una posición dentro del mismo.

```
...
////Firma y rúbrica/////
      string Rubrica="",F="";
      for (long long i=0; i<firma.size(); i=i+k){
            string bloque=firma.substr(i,k);
            ZZ signature=Expmod_rchino(Str2Num(bloque),d,n);
            F+= Completar0s(signature,k-1);
      }
      string R="";
      long long k2=Num2Str(nB).size();
      for (long long i=0; i<F.size(); i=i+k2){
            string bloque=F.substr(i,k2);
            ZZ
rubric=right2left_binary_modexp(Str2Num(bloque),eB,nB);
            R+=Completar0s(rubric,k2-1);
      }
      for (long long i=0; i<R.size(); i=i+num_dig){
            string bloque=R.substr(i,num_dig);
            long long pos=Str2Int(bloque);
            char a=alfabeto[pos];
            Rubrica+=a;
      }
      return Descifrado+" "+Rubrica;
      }
```

ALGORITMOS UTILIZADOS

a. Teorema del resto chino

- El teorema del resto chino, es un algoritmo que permite hallar una solución a un sistema de ecuaciones de congruencia modular, para el cual solo puede existir un X.

- **Base matemática:**

  Para la base matemática del resto chino, emplearemos un grupo de algoritmos/teoremas, entre los cuales está la división(módulo), el algoritmo de euclides, el algoritmo de euclides extendido y la inversa.

  Aquí supondremos que los números en los módulos son coprimos dos a dos, y por ende para cualquier grupo de enteros dados, hay un entero x que resuelve el sistema de congruencias con dichos números y módulos, si:

$$x \equiv num1 \bmod md1 \quad x \equiv num3 \bmod md3$$
$$x \equiv num2 \bmod md2 \quad x \equiv numk \bmod mdk$$

$$num_i \equiv num_j \,(mod\ euclides(md_i, md_j)$$

Como todos los módulos son coprimos entre sí, mediante el uso del algoritmo de euclides extendido (inversa) podemos asegurar lo siguiente:

$$N = md_1 md_2 \ldots md_k \; N_i = \frac{N}{md_i}$$
$$Existen\ dos\ enteros\ r_i\ y\ s_i\ que\ permiten: r_i md_i + s_i N_i = 1$$
$$s_i N_i \equiv 1 \bmod md_i \; s_i N_i \equiv 0 \bmod md_j$$

Lo cual nos permite definir la siguiente fórmula para hallar x:

$$x = \sum_{i=1}^{k} num_i s_i N_i = num_1 s_1 N_1 + num_2 s_2 N_2 + \ldots\ldots + num_k s_k N_k$$

Código C++:

```
ZZ Resto_Chino(ZZ* ai, ZZ* pi) {


      ZZ X = conv<ZZ>(0), P = conv<ZZ>(1);

      ZZ Pi[tam],qi[tam],x0[tam];


      if (Primos_Entre_Si(pi, ZZ(tam)) == 1) {

      for (int i = 0; i < tam; i++) {

            P *= pi[i];

      }


      for (int i = 0; i < tam; i++) {

            Pi[i] = P / pi[i];

            qi[i] = inversa(Pi[i], pi[i]);

      }
```

```
      for (int i = 0; i < tam; i++) {
            x0[i] = divi(ai[i] * Pi[i] * qi[i], P);
      }

      for (int i = 0; i < tam; i++) {
      X += x0[i];
      }
      X = divi(X, P);
      }
      return X;
}
ZZ pyq[2];
ZZ Expmod_rchino(ZZ b, ZZ e, ZZ m){
  ZZ a1=right2left_binary_modexp(b,divi(e,pyq[0]-1),pyq[0]);
  ZZ a2=right2left_binary_modexp(b,divi(e,pyq[1]-1),pyq[1]);
  ZZ as[]={a1,a2};
  ZZ mods[]={pyq[0],pyq[1]};
  ZZ mod = Resto_Chino(as,mods);
  return mod;
}
```

b. Miller-Rabin (test de primalidad)

Código C++:

```
bool miller_rabin(ZZ n,long t){
      if (n<2) return false;
```

```
	if (n==2 || n==3) return true;
	//if(par(n)) return false;//ahorrar trabajo si es par
	ZZ a,y,i,j,s,r;
	s=0;
	r=n-1;
	while(r>1 && par(r)){
	r>>=1;
	s++;
	}
	for(i=1;i<=t;i++){
	a=ZZ(2)+divi(seed_generator(bitz), n-3);
	y=right2left_binary_modexp(a,r,n);
	if(y!=1 && y!=n-1)
	{
		j=1;
		while(j<=s-1 && y!=n-1)
		{
			y=divi(y*y,n);
			if(y==1)return false;
			j++;
		}
		if(y!=n-1)return false;
	}
	}
	return true;
}
```

3. FUNCIONES ADICIONALES

a. Paridad

```
bool par(ZZ n){
    ZZ r=n-((n>>1)<<1);
    if (r==0) return true;
    return false;
}
```

b. Exponenciación base 2

```
ZZ exponenciacion(long e){
```

```
    if (e==0) return ZZ(1);
  ZZ A=conv<ZZ>(1);
    return A<<e;
}
```

c. Módulo

```
ZZ divi(ZZ a, ZZ n){
  ZZ r=a-((a/n)*n);
  if (r<0){
      if (n<0)
          return r-n;
  return r+n;
 }
   return r;
}
```

d. Completar datos (crear una cadena de 10*digs caracteres)

```
string Completar_datos(string d, int digs){
    int n_chars = 10 * digs;
    while (d.size()<n_chars){
        d+=d;
    }
    return d;
}
```

e. Criba

```
vector<long long> criba3(long long n) {
vector<long long> primes3;
primes3.push_back(2);
primes3.push_back(3);
for (long long i=5; i<n; i=i+2) {if (divi(i,3)!=0)
primes3.push_back(i);}
for (long long i=2; (primes3[i]*primes3[i])<n; i++) {
for (long long k=i+7; k<primes3.size(); k++) {if
(divi(primes3[k],primes3[i])== 0){
primes3.erase(primes3.begin()+k);primes3.resize(primes3.
size());primes3.shrink_to_fit();}
}
}
return primes3;
}
```

f. Primos entre sí

```
const int tam = 2;
bool Primos_Entre_Si(ZZ* Primos, ZZ tam) {

    for (int i = 0; i < tam; i++) {

        for (int j = i + 1; j < tam; j++) {
            if (Euclides(Primos[i], Primos[j]) != 1 ||
Primos[i] == Primos[j]) {
```

```
                    //cout << Primos[i] << " y " <<
            Primos[j] << " no son primos relativos" << endl;
                    return false;
                }
            }
        }

        return true;
    }
```

4. ALGORITMO RSA

Esta es la visualización del rsa con los algoritmos utilizados:

```
class RSA {
  ZZ d,p,q,oN;
  public:
  ZZ e,n,eB,nB;
  RSA(long bits){
     Generar_claves(bits);
     cout<<"p:"<<p<<" q:"<<q<<" n:"<<n<<" oN:"<<oN<<" e:"<<e<<"
d:"<<d<<endl;
  }
  RSA(ZZ e, ZZ n){
     this->eB=e;
     this->nB=n;
  }
  string cifrado(string m)
     {   ostringstream c,r;
     long num_dig=Num2Str(alfabeto.size()-1).size();
     long k=Num2Str(nB).size()-1;
     ////cifrado mensaje////
     while(divi((m.size()*num_dig),k)!=0)
          m+=' ';
     for (long i=0; i<m.size(); i++){
     long l=alfabeto.find(m[i]);
     c<<Completar0s(l,num_dig);
     }
     string C=c.str();
     string Cifrado="";
     for (long i=0; i<C.size(); i=i+k){
          string bloque=C.substr(i,k);
          ZZ
cypher=right2left_binary_modexp(Str2Num(bloque),eB,nB);
          Cifrado+= Completar0s(cypher,k+1);
     }
     /////rúbrica y firma/////
     long k2=Num2Str(n).size()-1;
```

```
      string d_F=Completar_datos(datos,k2);
      while(divi((d_F.size()*num_dig),k2)!=0 ||
divi(((k2+1)*((d_F.size()*num_dig)/k2)),k)!=0){
            d_F+=' ';}
      for (long i=0; i<d_F.size(); i++){
            long l=alfabeto.find(d_F[i]);
            r<<Completar0s(l,num_dig);
      }
      string R=r.str();
      string Rubrica="";
      for (long i=0; i<R.size(); i=i+k2){
            string bloque=R.substr(i,k2);
            ZZ rubric=right2left_binary_modexp(Str2Num(bloque),d,n);
            Rubrica+= Completar0s(rubric,k2+1);
      }
      string Firma="";
      for (long i=0; i<Rubrica.size(); i=i+k){
            string bloque=Rubrica.substr(i,k);
            ZZ
signature=right2left_binary_modexp(Str2Num(bloque),eB,nB);
            Firma+= Completar0s(signature,k+1);
      }
      return Cifrado+" "+Firma;
      }
  string descifrado(string m)
      {   long p = m.find(" ");
      string msje= m.substr(0,p);
      string firma=m.substr(p+1,m.size()-1);
      /////Descifrado//////
      string Descifrado="",D="";
      long long num_dig=Num2Str(alfabeto.size()-1).size();
      long long k=Num2Str(n).size();
      for (long long i=0; i<msje.size(); i=i+k){
            string bloque=msje.substr(i,k);
            ZZ decode=Expmod_rchino(Str2Num(bloque),d,n);
            D+= Completar0s(decode,k-1);
      }
      for (long long i=0; i<D.size(); i=i+num_dig){
            string bloque=D.substr(i,num_dig);
            long long pos=Str2Int(bloque);
            char a=alfabeto[pos];
            Descifrado+=a;
      }
      ////Firma y rúbrica/////
      string Rubrica="",F="";
      for (long long i=0; i<firma.size(); i=i+k){
            string bloque=firma.substr(i,k);
            ZZ signature=Expmod_rchino(Str2Num(bloque),d,n);
```

```
            F+= Completar0s(signature,k-1);
      }
      string R="";
      long long k2=Num2Str(nB).size();
      for (long long i=0; i<F.size(); i=i+k2){
            string bloque=F.substr(i,k2);
            ZZ
rubric=right2left_binary_modexp(Str2Num(bloque),eB,nB);
            R+=Completar0s(rubric,k2-1);
      }
      for (long long i=0; i<R.size(); i=i+num_dig){
            string bloque=R.substr(i,num_dig);
            long long pos=Str2Int(bloque);
            char a=alfabeto[pos];
            Rubrica+=a;
      }
      return Descifrado+" "+Rubrica;
      }
  void Generar_claves(long bits){
      bitz=ZZ(bits);
      exp_bitsminus1=exponenciacion(bits-1);
      expbits=exp_bitsminus1<<1;
      expbits_minus1=expbits-1;
      intervalo=expbits_minus1-exp_bitsminus1+1;
      pyq[0]=p=GenPrimo(bits);//Str2Num("");
      pyq[1]=q=GenPrimo(bits);//Str2Num("");
      while (p==q){
      q=GenPrimo(bits);
      }
      n=p*q;//Str2Num("");
      oN=(p-1)*(q-1);
      e=divi(random_numberBBS(bitz),oN);//Str2Num("");
      ZZ ee=BinaryGCD(e,oN);
      while(ee!=1){
      e=divi(random_numberBBS(bitz),oN);
      ee=BinaryGCD(e,oN);
      }
      d=inversa_1(e,oN);//Str2Num("");
  }
};
```

4. CONCLUSIONES GENERALES

Como conclusión final nuestro algoritmo RSA es eficiente, ya que logra cifrar y descifrar sin inconvenientes (ej: retornar caracteres basura) mensajes y firmas de distintas longitudes -la mayor cantidad de páginas cifradas fue de 105- y con claves de hasta 2048 bits. Además, hace uso de teoremas como el Resto Chino para hacer más veloz el proceso de encriptación y desencriptación. Y, aunque el tiempo de generación de claves puede ser muy variado, el método que utilizamos, haciendo uso de 2 semillas y distintos algoritmos para generar números pseudoaleatorios, hace que los números generados sean criptográficamente seguros.

5. REFERENCIAS

[01] Handbook of Applied Cryptography, Menezes, Oorschot, Vanstone. CRC Press, New York, fifth edition (2001). http://www.cacr.math.uwaterloo.ca/hac/

[02] A computational introduction to Number Theory and Algebra. Victor Shoup. http://www.shoup.net/ntb/ntb-v2.pdf

[03] Numerical Recipes in C : The Art of Scientific Computing. William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling

[04] Chapter 10. Number theory and Cryptography. https://silo.tips/download/chapter-numbertheory-and-cryptography-contents

[05] A comparison of several greatest common divisor (GDC) Algorithms. https://www.ijcaonline.org/volume26/number5/pxc3874253.pdf

[06] Introducción a la Teoría de Números. Ejemplos y algoritmos. Walter Mora https://repositoriotec.tec.ac.cr/bitstream/handle/2238/6299/introducci%C3%B3nteor%C3%ADa-n%C3%BAmeros.pdf?sequence=1&isAllowed=y

6. COMENTARIOS

Enlace GitHub: https://github.com/RodATS/Rodrigo_Torres_Sotomayor.git