INFORME RSA

Apellidos y Nombres de los integrantes del grupo

- Apaza Coaquira Eileen Karin
- Gallegos Vilca Renzo Leonardo
- Guillen Davila Marco Antonio
- Villagómez Iquira Angel

Porcentaje de participación de cada uno para la elaboración del código (0-100%)

- Apaza Coaquira Eileen Karin (23%)
- Gallegos Vilca Renzo Leonardo(30%)
- Guillen Davila Marco Antonio(23%)
- Villagómez Iquira Angel(23%)

Consideraciones:

El responsable del grupo debe de responder a este correo con el link del repositorio del github donde esté la carpeta del código. Anexe el informe en formato pdf.

Fecha de entrega: jueves 08 de julio 11:00 a.m.

De acuerdo al envío de sus trabajos, se programará las reuniones para la evaluación del código, el día martes de las 13:45 a 15:00 hras. Tiempo máximo de evaluación por grupo: 15 minutos.

Explicar y colocar el código que usaron para el algoritmo de criptografía RSA con firma digital, siguiendo la estructura que se les pide a continuación:

1. Estructura del main()

```
int main() {

    string msg = "comehere#";
    int opt;

    cout << "Choose an option" << endl;
    cout << right << setw(12)
         << "Encode (1)" << endl;
    cout << right << setw(12)
         << "Decode (2)" << endl;
    cout << "Option: ";
    cin >> opt;
    cin.ignore();

    if ((opt != 1) && (opt != 2)) {
```

```
        return 0;
    }

    // Objeto Receptor:
    // - Envía la clave pública e
    // - Envía el producto N
    // Directorio público
    RSA Receiver(3, 1003);
    // Objeto Emisor:
    // Lee el directorio público generado por el emisor
    RSA Emitter;

    if (opt == 1) {

        string code = Emitter.encode(msg);
        cout << "\nEncoded message: " << code;
    }
    else if (opt == 2) {

        string code = Receiver.decode(msg);
        cout << "\nDecoded message: " << code;
    }
    return 0;
}
```

2. Generación de claves

- **Generación de números aleatorios**

Para la generación de números aleatorios utilizamos Blum Blum Shub, este se utiliza como un generador de números pseudoaleatorios(es pseudo ya que no es un número verdaderamente aleatorio y su asignación al azar depende de un semilla aleatoria)
El constructor de la clase BBS tiene como parámetro (p,q seed) siendo variables tipo int y dentro del constructor se le asignan las variables y una operación para tener n que es resultado del producto de p*q , siendo p y q primos. En la función getRandNum(), se declara nextRandNum como una variable tipo int el cual será igual a mod((this->x0 * this->x0), this->n) , este siempre debe ser igual a 1, la función cumple con la fórmula :

$$x_{n+1} = (x_n)^2 \bmod M$$

```
// BBS (RNG)

class BBS {
private:
        int p, q, n, x0; // declaracion de variables en privado
```

```
public:
	BBS(int p, int q, int seed) {
		//mcd(p,q)=1
		  this->p = p;
		  this->q = q;
		  this->x0 = seed;
		  this->n = p * q;
	}
	// int getRandBit()
	// {
	//	return mod(this->getRandNum(),2);
	// }
	int getRandNum() {
		  int nextRandNum = mod((this->x0 * this->x0), this->n);
		  this->x0 = nextRandNum;
		  return nextRandNum;
	}
};

int binary_exponentiation(unsigned long long power) {
	int two = 2;
	two <<= power;

	return two;
}

int RANDBIGINTEGER(const unsigned long long BITS) {
	const unsigned long long TIME = time(NULL);
	const int MIN = binary_exponentiation(BITS - 1);
	const int MAX = binary_exponentiation(BITS);
	unsigned long long position = TIME;
	return 1;
}
```

- **Generación de primos:**

La función utilizada para la generación de primos hace uso del test de miller-rabin. En la función en la entrada ingresa un número natural n>1, el número k de veces que se ejecuta el test y nos determina la fiabilidad del test, de salida nos da COMPUESTO si n es compuesto y POSIBLE PRIMO si n es un posible primo.
El bucle repite k veces: elegir al azar en el rango [2, n - 2],
  x ← ad mod n
  si x = 1 o x = n - 1 entonces haz el siguiente BUCLE
  para r = 1 .. s - 1
    x ← x2 mod n
    si x = 1 entonces devuelve compuesto
    si x = n - 1 entonces continúa BUCLE
  retorno compuesto
volver probablemente primo

```
bool RSA::millerTest(int d, int n) {
	int a = 2 + rand() % (n - 4);
	int x = kary(a, d, n);
	if (x == 1 || x == n - 1)
		return true;

	while (d != n - 1) {
		x = (x * x) % n;
		d *= 2;
		if (x == 1)
			return false;
		if (x == n - 1)
			return true;
	}
	return false;
}
```

- **Algoritmo de Euclides**

Para esta función nos basamos en el algoritmo binario de MCD, como su nombre nos indica es un algoritmo que nos permite hallar el mcd de 2 números reemplazando las divisiones por shift aritméticos, comparaciones y restas. Primero seleccionamos 2 números a y b, si ambos números son 0, regresará como mcd=0. Si uno de los números es cero el mcd será el otro número pues todos los números dividen a cero. Si a y b son pares, estos serán divididos por shift por 1, lo cual significa que en sistema binaria se le quitara un cero de la derecha, lo cual equivale a la división por 2, junto con esto tendremos un contador k el cual nos permitirá saber el número de veces que se requirió para dividir por 2 esos números a la vez; pero sii “a” sigue siendo par entonces se le dividirá a la mitad hasta que sea impar. Una vez “a” sea completamente impar haremos lo mismo con “b”. Ambos siendo impar haremos que estos se resten, para que esto siempre si a llegase a ser mayor que a “b” estos tendrán que cambiar de valores y seguir restando hasta que “b” sea 0, esto significa que “a” será el mínimo valor divisible común de ambos números, aunque este aún no es mcd. Para hallar el valor de del mcd “a” se multiplicará con k el cual representa la cantidad de veces que ambos números se dividieron por 2, de esta manera obtenemos el mcd.

```
ZZ RSA::binary(ZZ a, ZZ b) {
	if (a == 0)
		return b;
	if (b == 0)
		return a;
	ZZ k;
	for (k = 0; ((a | b) & 1) == 0; ++k) {
		a >>= 1;
		b >>= 1;
	}
	while ((a & 1) == 0)
		a >>= 1;

	do {
		while ((b & 1) == 0)
			b >>= 1;
		if (a > b)
			swap(a, b);
```

```
            b = (b - a);
        } while (b != 0);
        a = a << k;
        return a;
}
```

- Inversa
  - ➢ **Euclides extendido:**

```
vector<ZZ> RSA::euclidExtendedAlgorithm(ZZ a, ZZ b)
{
   // Initialization
   ZZ r_1 = a;
   ZZ r_2 = b;
   ZZ s_1 = ZZ(1);
   ZZ s_2 = ZZ(0);
   ZZ t_1 = ZZ(0);
   ZZ t_2 = ZZ(1);

   while (r_2 != 0) {

      // Quotient
      ZZ quotient = r_1 / r;

      // Updating r
      ZZ temp = r;
      r = r_1 - quotient * r;
      r_1 = temp;

      // Updating s
      temp = s_2;
      s_2 = s_1 - quotient * s_2;
      s_1 = temp;

      // Updating t
      temp = t_2;
      t_2 = t_1 - quotient * t_2;
      t_1 = temp;
   }

   // Vector to store values
```

```
    vector<ZZ> result;

    // Greatest common divisor: result[0]
    result.push_back(r_1);
    // x: result[1]
    result.push_back(s_1);
    // y: result[2]
    result.push_back(t_1);

    return result;
}



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

ZZ RSA::inverse(ZZ a, ZZ b)
{
    ZZ s_1 = euclidExtendedAlgorithm(a, b)[1];
    if (s_1 < 0) {
        s_1 = modulus(s_1, b);
    }
    return s_1;
}
```

3. Formación de Bloques

- **Llenar ceros**

  Esta función recibe 2 valores, el tamaño y el string que queremos aumentar los 0, posteriormente de la diferencia de tamaño y el tamaño del string creamos un string de ceros (string zeros) que posteriormente se le aumentará el resto del string original. Por último el string original estar referenciado solo le damos el valor de zeros.

```
void addZeros(int size, string &block) {
    string zeros(size - block.size(), '0');
    zeros += block;
    block = zeros;
}
```

- **Convertir string a enteros (ZZ)**

  La función string2Num convierte tipo string a ZZ, se utiliza istringstream con el cual podemos el string puede ser transmitido y guardado usando el operador de extracción (>>). El operador de extracción leerá hasta que se alcance el espacio en blanco o hasta que falle la secuencia.
  Por último retornamos nuestros enteros que en este caso son numberZZ.

```
ZZ string2Num(string message){
    ZZ numberZZ;
    istringstream num(message);
    num >> numberZZ;
    return numberZZ;
}
```

- **ZZ a string**

  La función num2String convierte las variables tipo int a string, se utiliza ostringstream que es una clase de flujo de salida para operar cadenas.

```
string num2String(ZZ number) {
    ostringstream stringZZ;
    stringZZ << number;
    return stringZZ.str();
}
```

- **Dividir bloques**

```
string mensaje_antes;
    int tamano_mensaje = mensaje.length();

    string tamano_alfabeto = num2String(alphabet.length());
    int tamano = tamano_alfabeto.length();

    for(int i = 0; i < tamano_mensaje; i++) {
        int posicion = alphabet.find(mensaje[i]);
```

```
    string tamano_posicion = num2String(posicion);
    int tamano_pos = tamano_posicion.length();

    if( tamano_pos < tamano) {
    string pos = num2String(posicion);
    agregar_ceros(tamano, pos);
    mensaje_antes += pos;
    }
    else {
    string pos = num2String(posicion);
    mensaje_antes += pos;
    }
}
```

4. Exponenciación modular:

Para la exponenciación modular utilizamos la función basada en el algoritmo K-ary. Lo primero que hacemos en la función es inicializar 3 variables: a (x), b (n) , c(m) y res=1, después verificamos el módulo de x en función de m, dando ese valor a “a”. Usaremos como referencia el exponente para saber cuantas veces usaremos nuestra función, pero para esto iremos dividiendo entre 2 hasta que llegue a 0. Si es nuestro exponente es negativo entonces hallaremos el módulo res*a de c, de esta manera iremos hallando el módulo de cada exponente uno por uno. “a” deberá elevarse al cuadrado y después aplicarle módulo b, esto se hará debido a que usualmente para hallar el módulo de un número con exponente se le realiza constantemente a sus valores por separado la función módulo. Se realizan los pasos anteriores constantemente hasta que b llegue a cero y el programa nos retorne nuestro resultado (res).

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

ZZ binaryEuclidAlgorithm(ZZ a, ZZ b) {
    ZZ g = ZZ(1);
    while ((modulus(a, ZZ(2)) == 0) && (modulus(b, ZZ(2)) == 0)) {
```

```
        a = a / 2;
        b = b / 2;
        g = 2 * g;
    }
    while (a != 0) {
        if (modulus(a, ZZ(2)) == 0) {
            a = a / 2;
        }
        else if (modulus(b, ZZ(2)) == 0) {
            b = b / 2;
        }
        else {
            ZZ t = abs(a - b) / 2;
            if (a >= b) {
                a = t;
            }
            else {
                b = t;
            }
        }
    }
    return g * b;
}

ZZ phi(ZZ modulus)
{
    ZZ result = ZZ(1);
    for (ZZ i = ZZ(2); i < n; i++)
        if (binaryEuclidAlgorithm(i, ZZ(modulus)) == ZZ(1))
            result++;
    return result;
}

int RSA::kary(int base, int exponent, int modulus) {
    if(binaryEuclidAlgorithm(base, modulus) == ZZ(1)){
        ZZ phiModulus = phi(m);
        exponent = modulus(exponent, phiModulus);
    }
    base = modulus(base, modulus);
    ZZ result = conv<ZZ>("1");
    while (exponent > 0) {
```

```
        if ((exponent & 1) == 1)
        result = modulus(result * base, modulus);
        base = modulus(base * base, modulus);
        exponent >>= 1;
    }
    return result;
}
```

5. Función de cifrado

```
string RSA::encode(string mensaje) {

    string respuesta;
    string mensaje_antes = dividir_bloques(mensaje, alphabet);

    //cout << "Mensaje en bloques: " << mensaje_antes << endl;

    int bloque = num2String(N).length() - 1;
    int longitud = mensaje_antes.length();

    for(int i = 0; i < longitud; i += bloque){
        int base = 0;
        if(i + bloque > longitud) {
            base = string2Num(mensaje_antes.substr(i, longitud - i));
        }
        else {
            base = string2Num(mensaje_antes.substr(i, bloque));
        }

        int exp = kary(base, e, N);
        string tamano_resultado = num2String(exp);
        int tamano_res = tamano_resultado.length();

        string N_str = num2String(N);
        int tam_N = N_str.length();
        if( tamano_res < tam_N) {
            string pos = num2String(exp);
            agregar_ceros(tam_N, pos);
            respuesta += pos;
        }
        else {
            string pos = num2String(exp);
            respuesta += pos;
        }

    }
    return respuesta;
}
```

6. Función de descifrado

```
string RSA::decode(string mensaje) {

    string respuesta;

    string N_str = num2String(N);
    int tam_N = N_str.length();
    int longitud = mensaje.length();
    int tamano = 0;

    for(int i = 0; i < longitud; i += tam_N){
        int base = 0;
        if(i + tam_N > longitud) {
            base = string2Num(mensaje.substr(i, longitud - i));
        }
        else {
            base = string2Num(mensaje.substr(i, tam_N));
        }

        int exp = kary(base, e, N);

        string tamano_resultado = num2String(exp);
        int tamano_res = tamano_resultado.length();

        string tamano_alfabeto = num2String(alphabet.length());

        tamano = tamano_alfabeto.length();

        if( tamano_res < tam_N - 1) {
            string pos = num2String(exp);
            agregar_ceros(tam_N - 1, pos);
            respuesta += pos;
        }
        else {
            string pos = num2String(exp);
            respuesta += pos;
        }

    }
    string conv = convert(respuesta, alphabet, tamano);
    return conv;
}
```

7. Firma digital

Nota: Si en una función llama a otras funciones, adicionales dentro del ítem que le corresponda. Puede anexar otra documentación que considere necesaria para la revisión del algoritmo.