Criptografía y Computación
Práctica 1

Miguel Ángel Fernández Alonso
DNI: 76065289B

Índice

Introducción

Esta memoria consta de una breve descripción del código (como es pedido en la entrega de Prado) junto a una captura de pantalla de las gráficas de los tiempos obtenidos.

Ejercicio 1

Implementa, a partir de un tipo de dato entero sin signo de tamaño fijo (preferentemente de 64 bits), un tipo de dato que permita representar números enteros de tamaño arbitrario.

Este tipo de dato ha sido implementado en C++ implementado con una clase llamada “BigNumber”.

```
    IntegerArbitrario(const std::vector<uint64_t>& digits) : digits(digits) {
    // Eliminar ceros no significativos al principio
        this->digits = digits;
        while (!this->digits.empty() && this->digits.back() == 0) {
            this->digits.pop_back();
        }
}
```

Presenta de constructor para usar vectores.
Las funciones implementadas son para imprimir el número, para convertir a cadena en formato hexadecimal y para convertir desde cadena en formato hexadecimal.

```cpp
//Función para imprimir el número
void printNumber() const{
    for(auto it = digits.rbegin(); it != digits.rend(); ++it){
        std::cout << *it;
    }
    std::cout << std::endl;
}


// Método para convertir a cadena en formato hexadecimal
std::string toHexString() const{
    std::stringstream ss;

    if(digits.size() == 1 && digits[0] == 0){
        //El número es cero
        ss << "0x0";
    }
    else{
        //Determinamos el signo
        bool isNegative = (digits.size() > 0 && digits.back() >= 0x8000000000000000);

        //Mostrar el prefijo 0x o -0x
        ss << ((isNegative) ? "-0x" : "0x");

        //Mostrar los digitos en formato hexadecimal
        for(auto it = digits.rbegin(); it != digits.rend(); ++it){
            ss << std::setfill('0') << std::setw(16) << std::hex << *it;
        }
    }

    return ss.str();

}
```

```cpp
//Método para convertir desde cadena en formato hexadecimal
static IntegerArbitrario fromHexString(const std::string& hexString){
    size_t startIndex = (hexString[0] == '-') ? 3 : 2; // Considerar el signo y el prefijo 0x

    std::vector<uint64_t> resultDigits;
    for(size_t i = hexString.size() - 1; i >= startIndex; --i){
        uint64_t digit;
        std::stringstream ss;
        ss << std::hex << hexString[i];
        ss >> digit;
        resultDigits.push_back(digit);
    }

    //Eliminamos ceros no significativos al principio
    while(resultDigits.size() > 1 && resultDigits.back() == 0){
        resultDigits.pop_back();
    }

    return IntegerArbitrario(resultDigits);
}
```

He obviado el uso de un destructor para liberar memoria al usar en este apartado vectores.

Ejercicio 2

1. Implementa la operación cambio de signo, que para un elemento del tipo de dato creado que represente el número entero a devuelva uno que represente −a.

La clase usada a partir de ahora es similar a la del ejercicio anterior. Presenta constructores, un vector de enteros que contenga los dígitos, la base en formato “uint64_t” y un booleano para comprobar que el número no sea negativo.

```
private:
    std::vector<int> digits; //Lista de dígitos
    bool isNegative;        //Indica si el número es negativo
    uint64_t base;

public:
    //Constructor por defecto
    BigNumber() : isNegative(false){
        digits.push_back(0);
        base = uint64_t(pow(2, 64));
    }


    // Constructor que toma un vector de dígitos y un booleano que indica el signo
    BigNumber(const std::vector<int>& digits, bool isNegative) : digits(digits), isNegative(isNegative) {}
```

```
    //Constructor que recibe un entero inicial
    BigNumber(int num){
        isNegative = (num < 0);
        num = std::abs(num);

        while(num > 0){
            digits.push_back(num % base);
            num /= base;       //Base 2^64
        }

        if(digits.empty()){
            digits.push_back(0);    // Si el número es cero, al menos debe tener un dígito
        }
    }
```

La función para cambio de signo:

```
//Función para cambiar el signo del número
BigNumber negate() const {
    BigNumber result = *this;   //Creamos una copia del número actual
    result.isNegative = !result.isNegative; //Invertimos el signo
    return result;
}
```

2. Implementa las operaciones suma y resta de dos elementos del tipo de dato creado, siguiendo el algoritmo escolar de la suma.

Para implementar esto he seguido el pseudocódigo proporcionado en la lección 1.

```
//Función para sumar dos números
BigNumber add(const BigNumber& other) const{
    BigNumber result;
    int carry = 0;

    size_t maxSize = std::max(digits.size(), other.digits.size());

    for(size_t i = 0; i < maxSize || carry; ++i){
        int digitA = (i < digits.size()) ? digits[i] : 0;
        int digitB = (i < other.digits.size()) ? other.digits[i] : 0;

        auto sumResult = sumDigits(digitA, digitB, carry);

        result.digits.push_back(sumResult.first);
        carry = sumResult.second;
    }

    result.isNegative = isNegative; //Conservamos el signo del primer número
    return result;
}
```

```cpp
BigNumber operator-(BigNumber& other){
    BigNumber result;
    int borrow = 0;

    size_t maxSize = std::max(digits.size(), other.digits.size());

    for(size_t i = 0; i < maxSize; ++i){
        int digitA = (i < digits.size()) ? digits[i] : 0;
        int digitB = (i < other.digits.size()) ? other.digits[i] : 0;

        if(digitA >= digitB + borrow){
            result.digits.push_back(digitA - digitB - borrow);
            borrow = 0;
        }
        else{
            result.digits.push_back(base + digitA - digitB - borrow);
            borrow = 1;
        }
    }

    //Eliminamos ceros no significativos
    while(result.digits.size() > 1 && result.digits.back() == 0){
        result.digits.pop_back();
    }

    //Determinarmos el signo del resultado
    if(borrow == 1){
        result.isNegative = true;
    }
    else{
        result.isNegative = isNegative;     //Conservamos el signo del primer número
    }

    return result;
}
```

A partir de ahora he tenido que sobrecargar operadores para usar los algoritmos en lugar de operadores lógicos.

Ejercicio 3.

Implementa la operación producto de dos elementos del tipo de dato creado, siguiendo el algoritmo escolar para la multiplicación.

He implementado la siguiente función auxiliar para poder implementar el algoritmo escolar de la multiplicación.

```cpp
//Función auxiliar para multiplicar dos dígitos (Algoritmo 5)
static std::pair<int, int> multiplyDigits(int a, int b, int gamma) {
    int c = a * b + gamma;
    int gamma_star = 0;
    uint64_t base = pow(2, 64);
    while(c >= base){ //Mayor o igual que la base
        c -= base;
        gamma_star++;
    }

    return std::make_pair(c, gamma_star);
}
```

```cpp
class BigNumber{
    BigNumber operator*(const BigNumber & other) const{
        BigNumber result;
        BigNumber tempResult;

        for(size_t j = 0; j < other.digits.size(); ++j){
            int gamma = 0;
            tempResult.digits.clear();

            for(size_t i = 0; i < digits.size(); ++i){
                auto product = multiplyDigits(digits[i], other.digits[i], gamma);
                tempResult.digits.push_back(product.first);
                gamma = product.second;
            }

            while(gamma > 0){
                tempResult.digits.push_back(gamma % base);
                gamma /= base;
            }

            //Agregar ceros al final según el lugar que ocupa en la multiplicación
            for(size_t k = 0; k < j; ++k){
                tempResult.digits.insert(tempResult.digits.begin(), 0);
            }

            result = result.add(tempResult);
        }

        //Determinamos el signo del resultado
        if(isNegative != other.isNegative){
            result.isNegative = true;
        }
        else{
            result.isNegative = false;
        }

        return result;
```

Sobrecargando de nuevo el operador de la multiplicación.

Ejercicio 4

Ejercicio 4. Implementa la operación división, que devuelva tanto el cociente como el resto, de dos elementos del tipo de dato creado, siguiendo el algoritmo escolar para la división. Recordemos que en la descripción que hemos hecho el resto siempre es positivo.

He implementado la siguiente función auxiliar para comprobar los valores absolutos de dos números. He hecho esto para poder comprobar que el dividendo sea igual o mayor que el divisor.

```
//Función para comparar los valores absolutos de dos números
int compareAbsoluteValues(const BigNumber& other) const{
    if(digits.size() > other.digits.size()){
        return 1;
    }
    else if(digits.size() < other.digits.size()){
        return -1;
    }

    for(int i = digits.size() - 1; i >= 0; --i){
        if(digits[i] > other.digits[i]){
            return 1;
        }
        else if(digits[i] < other.digits[i]){
            return -1;
        }
    }

    return 0;
}
```

```cpp
std::pair<BigNumber, BigNumber> divide(const BigNumber & divisor) const{
    if(divisor.digits.size() == 1 && divisor.digits[0] == 0){
        throw std::invalid_argument("División por cero");
    }

    BigNumber quotient;
    BigNumber remainder = *this;    //Copia del dividendo

    //Preparamos el divisor para compararlo con el dividendo
    BigNumber tempDivisor;
    for(size_t i = 0; i < remainder.digits.size(); ++i){
        tempDivisor.digits.push_back(0);
    }
    tempDivisor.digits.push_back(divisor.digits[0]);

    //Mientras el dividendo sea igual o mayor que el divisor
    while(remainder.compareAbsoluteValues(tempDivisor) != -1){
        quotient.digits.push_back(0);   //Incrementamos el cociente

        //Realizamos la resta para obtener el nuevo dividendo
        remainder = remainder - tempDivisor;

        //Preparamos el nuevo divisor
        tempDivisor.digits.pop_back();
        tempDivisor.digits.insert(tempDivisor.digits.begin(), 0);
    }


    //Determinar el signo del cociente y del resto
    quotient.isNegative = (isNegative != divisor.isNegative);
    remainder.isNegative = isNegative;

    return std::make_pair(quotient, remainder);
}
```

Ejercicio 5

Implementa la operación producto de dos elementos del tipo de dato creado, siguiendo el algoritmo de Karatsuba.

```cpp
//Función para multiplicar usando karatsuba
static BigNumber karatsuba(const BigNumber& num1, const BigNumber& num2){
    size_t m = std::max(num1.digits.size(), num2.digits.size());

    //Caso base
    if(m == 1){
        auto product = multiplyDigits(num1.digits[0], num2.digits[0], 0);
        std::vector<int> digits = {product.first};
        if(product.second > 0){
            digits.push_back(product.second);
        }

        return BigNumber(digits, false);
    }

    size_t halfSize = m / 2;

    //Dividir los números en partes
    BigNumber a0(std::vector<int>(num1.digits.begin(), num1.digits.begin() + halfSize), false);
    BigNumber a1(std::vector<int>(num1.digits.begin() + halfSize, num1.digits.end()), false);
    BigNumber b0(std::vector<int>(num2.digits.begin(), num2.digits.begin() + halfSize), false);
    BigNumber b1(std::vector<int>(num2.digits.begin() + halfSize, num2.digits.end()), false);

    //Calcular partes recursivamente
    BigNumber c0 = karatsuba(a0, b0);
    BigNumber c2 = karatsuba(a1, b1);
    BigNumber c1 = karatsuba(auxKaratsuba(a0, a1), auxKaratsuba(b0, b1));
    c1 = subtractKaratsuba(c1, c0);
    c2 = subtractKaratsuba(c1, c2);

    //Ajustar el tamaño del resultado
    std::vector<int> zeros(2 * halfSize, 0);
    c0.digits.insert(c0.digits.end(), zeros.begin(), zeros.end());
    c1.digits.insert(c1.digits.end(), zeros.begin(), zeros.begin() + halfSize);

    //Sumar los resultados
    BigNumber result = auxKaratsuba(c0, c1);
    result = auxKaratsuba(result, c2);

    return result;

}
```

Tanto “auxKaratsuba” como “subtractKaratsuba” son funciones auxiliares de suma y resta simples para poder sumar y restar números grandes.

```cpp
// Función auxiliar para sumar dos números grandes
   static BigNumber auxKaratsuba(const BigNumber& num1, const BigNumber& num2) {
       std::vector<int> result;
       int carry = 0;
       int i = num1.digits.size() - 1;
       int j = num2.digits.size() - 1;
       uint64_t base = uint64_t(pow(2, 64));

       while (i >= 0 || j >= 0 || carry) {
           int sum = carry;
           if (i >= 0) {
               sum += num1.digits[i];
               i--;
           }
           if (j >= 0) {
               sum += num2.digits[j];
               j--;
           }

           result.push_back(sum % base);
           carry = sum / base;
       }

       std::reverse(result.begin(), result.end());
       return BigNumber(result, false);
   }
```

```cpp
   // Función auxiliar para restar dos números grandes
   static BigNumber subtractKaratsuba(const BigNumber& num1, const BigNumber& num2) {
       std::vector<int> result;
       int borrow = 0;
       int i = num1.digits.size() - 1;
       int j = num2.digits.size() - 1;
       uint64_t base = uint64_t(pow(2, 64));

       while (i >= 0) {
           int diff = num1.digits[i] - borrow;
           if (j >= 0) {
               diff -= num2.digits[j];
               j--;
           }
           if (diff < 0) {
               diff += base;
               borrow = 1;
           } else {
               borrow = 0;
           }
           result.push_back(diff);
           i--;
       }

       std::reverse(result.begin(), result.end());
       return BigNumber(result, false);
   }
```

La estructura de ambas funciones es igual.

Ejercicio 6

Implementa el Algoritmo Extendido de Euclides para dos elementos a, b del tipo de dato creado, que devuelva d, u, v de forma que d es el máximo común divisor de a y b y que d = au + bv.

Para este algoritmo he tenido que usar otro algoritmo de la división al pedido en el ejercicio anterior 4, ya que usaba otro algoritmo de división.
El algoritmo en cuestión es el siguiente:

```
//Función de división. Algoritmo 8
static std::pair<BigNumber, BigNumber> division(const BigNumber &a, const BigNumber &b){
    uint64_t base = uint64_t(pow(2, 64));
    //Verificamos si el divisor tiene solo un dígito
    if(b.digits.size() == 1){
        int divisor = b.digits[0];
        BigNumber quotient;
        std::vector<int> remainder = {0}; //Inicializamos a cero

        for(int j = a.digits.size() - 1; j >= 0; --j){
            int dividend = remainder[0] * base + a.digits[j]; //Parte entera de la división
            int qj = dividend / divisor;    //Cociente
            quotient.digits.push_back(qj);
            int rj = dividend % divisor;    //Residuo
            remainder[0] = rj;
        }

        std::reverse(quotient.digits.begin(), quotient.digits.end());
        return std::make_pair(quotient, BigNumber(remainder, false));
    }

    //Normalización
    int d = base / (b.digits.back() + 1);
    BigNumber normalizedA = multiplySingleDigit(a, d);
    BigNumber normalizedB = multiplySingleDigit(b, d);

    //Dividimos
    int n = normalizedA.digits.size();
    int m = normalizedB.digits.size();
    std::vector<int> quotientDigits(n - m + 1, 0);  //Inicializamos los dígitos del cociente

    for(int j = n; j >= m; --j){
        int qHat;
        if(normalizedA.digits[j] == normalizedB.digits[m - 1]){
            qHat = 0;
        }
        else{
            int dividend = normalizedA.digits[j] * base + normalizedA.digits[j - 1];
            qHat = dividend / normalizedB.digits[j - 1];
            if(qHat * normalizedB.digits[m - 2] >
                (dividend - qHat * normalizedB.digits[m - 1]) * base + normalizedA.digits[j - 2])
                    qHat--;
                }
        }

        //Restamos qHat * b de a
        BigNumber product = multiplySingleDigit(normalizedB, qHat);
        while(compare(normalizedA, j - m + 1, product)){
            qHat--;
            product = subtractAux(normalizedA, j - m + 1, product);
        }
```

```
        //Actualizamos el cociente
        quotientDigits[j - m] = qHat;
        normalizedA = subtractAux(normalizedA, j - m + 1, product);
    }

    std::reverse(quotientDigits.begin(), quotientDigits.end());
    auto divisionResult = BigNumber::division(normalizedA, BigNumber(d));
    BigNumber quotient = divisionResult.first;
    BigNumber remainder = divisionResult.second;

    return std::make_pair(quotient, remainder);
}
```

He usado una función auxiliar para poder calcular el máximo común divisor.

```
//Función para obtener el máximo común divisor utilizando el algoritmo de Euclides
static int euclideanAlgorithm(int a, int b){
    while(b != 0){
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

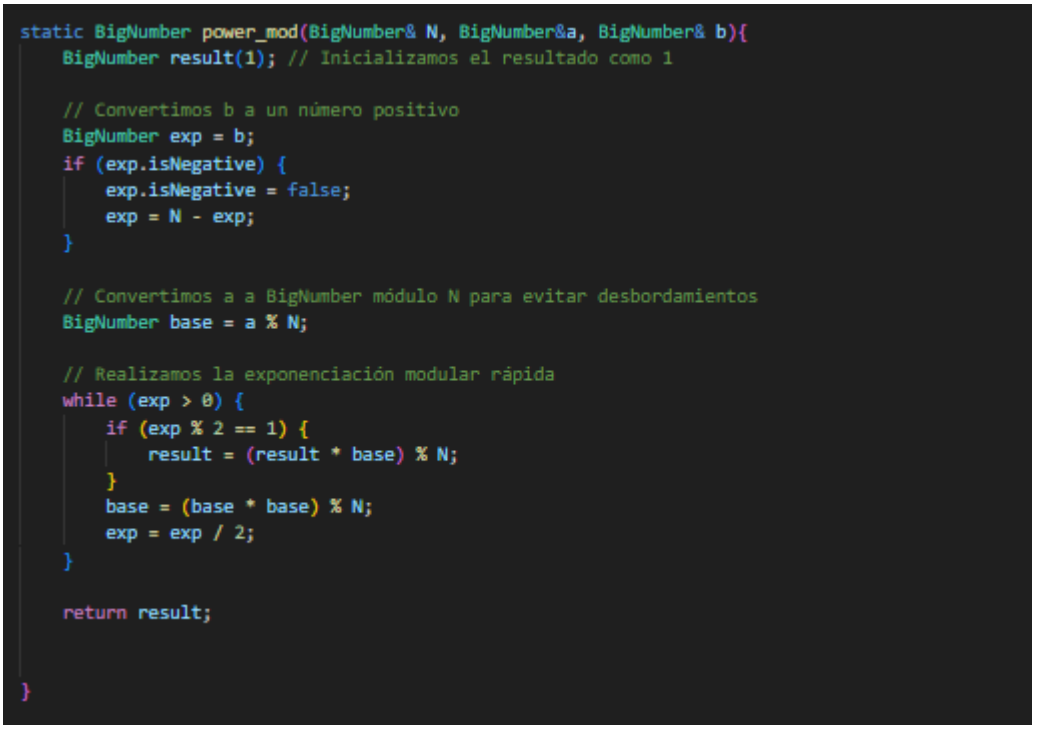
Para el algoritmo extendido de Euclides:

```cpp
//Algoritmo Extendido de Euclides. Algoritmo 10
static std::tuple<int, int, int> extendedEuclideanAlgorithm(BigNumber& a, BigNumber& b) {
    BigNumber m = a;
    BigNumber u0 = BigNumber({1}, false);
    BigNumber v0 = BigNumber({0}, false);
    BigNumber u1 = BigNumber({0}, false);
    BigNumber v1 = BigNumber({1}, false);

    while(b != BigNumber::zero()){
        auto divisionResult = BigNumber::division(a, b);
        BigNumber q = std::get<0>(divisionResult);
        BigNumber r = std::get<1>(divisionResult);

        a = b;
        b = r;

        BigNumber temp1 = q * u1;
        BigNumber temp2 = q * v1;
        BigNumber u = u0 - temp1;
        BigNumber v = v0 - temp2;
        u0 = u1;
        v0 = v1;
        u1 = u;
        v1 = v;
    }

    int gcd = euclideanAlgorithm(m.toInt(), a.toInt());
    int u = u0.toInt();
    int v = v0.toInt();

    return std::make_tuple(gcd, u, v);
}


//Sobrecarga del operador de exponenciación (^)
BigNumber power(int exponent) const{
    BigNumber result({1}, false);
    BigNumber base = *this;

    while(exponent > 0){
        if(exponent % 2 == 1){
            result = result * base;
        }
        base = base * base;
        exponent /= 2;
    }

    return result;
}
```

Ejercicio 7

Implementa la operación exponenciación modular rápida para tres elementos a, b y n del tipo de dato creado, de forma que compute a^b módulo n mediante cuadrados iterados.

El algoritmo en cuestión:



```
static BigNumber power_mod(BigNumber& N, BigNumber&a, BigNumber& b){
    BigNumber result(1); // Inicializamos el resultado como 1

    // Convertimos b a un número positivo
    BigNumber exp = b;
    if (exp.isNegative) {
        exp.isNegative = false;
        exp = N - exp;
    }

    // Convertimos a a BigNumber módulo N para evitar desbordamientos
    BigNumber base = a % N;

    // Realizamos la exponenciación modular rápida
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % N;
        }
        base = (base * base) % N;
        exp = exp / 2;
    }

    return result;


}
```

Todos los operadores han sido sobrecargados al estar trabajando con otro tipo de dato.

Ejercicio 8

Implementa el test de Miller-Rabin para un entero del tipo de dato creado.

```
//Test de Miller-Rabin Algoritmo 17
bool strongPseudoprime(BigNumber& n, BigNumber& a){
    if(BigNumber::gcd(a, n) != BigNumber({1}, false)){
        return false;
    }
    BigNumber temp = BigNumber({1}, false);
    BigNumber nMinusOne = n - temp;
    BigNumber t = nMinusOne;
    int s;
    while(t % BigNumber({2}, false) == BigNumber({0}, false)){
        t = t / BigNumber({2}, false);
        s++;
    }

    BigNumber b = a.power(t.toInt()) % n;
    if(b == BigNumber({1}, false) || b == nMinusOne){
        return true;
    }

    for(int r = 1; r < s; ++r){
        b = (b * b) % n;
        if(b == nMinusOne){
            return true;
        }
    }
    return false;
}

bool millerRabinTest(BigNumber& n, int k){
    BigNumber temp = BigNumber({1}, false);
    for(int i = 0; i < k; ++i){
        BigNumber a = getRandomNumber(BigNumber({2}, false), n - temp);
        if(!strongPseudoprime(n, a)){
            return false;
        }
    }
    return true;
}
```

El algoritmo está descompuesto en dos. Uno para comprobar si un número es pseudoprimo de otro y el propio test de Miller-Rabin.
Estos usan funciones tales como “gcd” la cual es la de obtener el máximo común divisor (forma euclídea vista antes), “power” el cual es el operador de exponenciación sobrecargado y una función básica para pasar a tipo de dato entero.

```
//Sobrecarga del operador de exponenciación (^)
BigNumber power(int exponent) const{
    BigNumber result({1}, false);
    BigNumber base = *this;

    while(exponent > 0){
        if(exponent % 2 == 1){
            result = result * base;
        }
        base = base * base;
        exponent /= 2;
    }

    return result;
}
```

Tiempos de ejecución

En el .zip entregado en la práctica se encontrará un archivo formato “.ods” en el cual podremos encontrar las gráficas con las medias y desviaciones típicas de los tiempos de ejecución al igual que con el tamaño de número “m” usado. He encontrado ciertos picos en las gráficas debido al alojamiento en memoria.