CIENCIA DE LA COMPUTACIÓN

ÁLGEBRA ABSTRACTA

TRABAJO DE INVESTIGACIÓN

- **Apaza Coaquira, Aarón Misash**
- **Choque Mayta, Gabriel Santiago**
- **Condori Gonzales, Jean Carlo**
- **Romero Guillen, Geraldo Mark**
- **Vilca Campana, José Enrique**

CCOMP3-1

2021

Los alumnos declaran haber realizado el trabajo de acuerdo a las normas De la universidad Católica San Pablo

Resumen:

Se implementaron las siguientes soluciones:

- *Binary exponentiation:*
  - *Right to left: Mediante el algoritmo de exponenciación bianria se realizan pequeñas y sucesivas potenciaciones que son divididas entre dos por cada llamada, siendo el ordende derecha hacia izquierda.*
  - *Left to right: Usa el mismo principio que el anterior solo que cambia el recorrido empezando por la izquierda hacia la derecha.*
- *Naive Exponentiation:* Por medio de un acercamiento binario(cadena de sumas de múltiplos de dos), y sumando la base una vez más si es necesario, calcula la potenciación.
- *Exponenciación modular rápida:* Es un algoritmo que se centra en dividir el exponente entre dos hasta que llegue a cero.
- *Teorema del Resto Chino:* Es un teorema que da una solución única a las congruencias lineales simultáneas con módulos coprimos
- *Exponenciación por mínima cadena de sumas:* Busca la mínima cadena de sumas de un exponente y la usa.

Criterios de Evaluación:

- Tiempo de ejecución respecto al número de bits.
- cálculo computacional

Algoritmo de mejor desempeño:

- Algoritmo del teorema resto chino

Introducción:

En la redacción resaltar el problema y objetivos de la investigación.
El presente trabajo de investigación está motivado por la necesidad de eficiencia y reducción de costo computacional en los algoritmos de exponenciación. Por consiguiente, todos los algoritmos presentados tienen como objetivo satisfacer este menester.

Por cada algoritmo en este trabajo de investigación se ahondará en su respectiva explicación detallada, implementación en pseudocódigo, acercamiento ejemplar al contenido de las variables en la implementación del algoritmo, convergencia y eficiencia respecto a bits y su fundamento matemático.

Contenido Teórico

Right to left binary exponentiation

Este algoritmo reduce drásticamente la cantidad de operaciones que se realiza en la exponenciación modular, este método incorpora la exponenciación binaria.

Lo cual se representa como una constante reducción del exponente por dos, representado por:

$$c \equiv \prod_{i=0}^{n-1} b^{a_i 2^i} \ (mod\ m)$$

Pseudo-Algoritmo:

```
long long binpow(long long a, long long b, long long m) {
    a %= m;
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res;
}
```

Seguimiento numérico:

$$3^8 mod\ 4$$

$$3^1 = 3$$
$$3^2 = (3^1)^2 = 3^2 = 9\ mod\ 4$$
$$3^4 = (3^2)^2 = 9^2 = 81\ mod\ 4$$
$$3^8 = (3^4)^2 = 81^2 = 6561\ mod\ 4$$

Implementación en C++:

```
ZZ Right_to_Left_Binary(ZZ g, ZZ e, ZZ m)
{
    ZZ A;
    A = 1;
    while (e != 0)
    {
        if ((e & 1) == 1)
            A = module(A * g, m);
        e >>= 1;
        g = module(g * g, m);
    }
    return A;
}
```

Left to right binary exponentiation

Tiene el mismo principio que el anterior algoritmo, la diferencia está en el recorrido que lleva de izquierda a derecha.

Pseudo-Algoritmo:

INPUT: g ∈ G and a positive integer e = (etet−1 ··· e1e0)2.
OUTPUT: ge.
A←1.
For i from t down to 0 do the following:
A←A · A.
If ei = 1, then A←A · g.
Return(A).

Seguimiento numérico:

$t = 8 \, and \, 283 = (100011011)_b$

i	8	7	6	5	4	3	2	1	0
e_i	1	0	0	0	1	1	0	1	1
A	g	g^2	g^4	g^8	g^{17}	g^{35}	g^{70}	g^{141}	g^{283}

Implementación en C++:

```
void left_to_right_binary(ZZ b, ZZ e, ZZ mod)
{
    ZZ A(1);
    string bin = toBinary(e);
    for (int i = bin.size(); i != -1; i--)
    {
        A = module(A * A, mod);
        if (bin[i] == '1')
        {
            A = module(A * b, mod);
        }
    }
    cout << A;
}
```

Naive Exponentiation

Definición:

Una de las consideraciones más importantes en cualquier algoritmo de exponenciación es evitar que los resultados intermedios sean demasiado grandes, este algoritmo toma el módulo en cada iteración para reducir el resultado constantemente.

Este algoritmo es también conocido como el método de **fuerza bruta** para la exponenciación modular, es correcto, pero no es muy eficiente, ya que se necesita un número de iteraciones igual al exponente para calcular la exponenciación modular.Con grandes exponentes este tiempo de ejecución es bastante lento : $O\ (p)$

Pseudo-Algoritmo:

$Input: Integers\ a, p, n$

$Output: r = a^{p}\ mod\ n$

$r \leftarrow 1$

$for\ i \leftarrow 1\ to\ p\ do$

$\quad r \leftarrow (r \times a)\ mod\ n$

$return\ r$

Seguimiento numérico: $4^{5} mod\ 497$

$$e = 10\ c = 4$$
$$e = 1\ c = 4\ mod\ 497 = 4$$
$$e = 2\ c = (4 \times 4)\ mod\ 497 = 16$$
$$e = 3\ c = (16 \times 4)\ mod\ 497 = 64$$
$$e = 4\ c = (64 \times 4)\ mod\ 497 = 256$$
$$e = 5\ c = (256 \times 4)\ mod\ 497 = 30$$

Implementación en C++:

```cpp
ZZ NaiveExponentiation(ZZ b, ZZ e, ZZ mod)
{
    ZZ r(1);
    for (ZZ i(1); i <= e; i++)
    {
        r = module(r * b, mod);
    }
    return r;
}
```

Exponenciación modular rápida

Definición:

Este algoritmo utiliza el método **repeated squaring,** que consiste en partir el exponente a la mitad constantemente :

$$b^4 = (b^2)^2 = b^2 \times b^2$$

Basado en una observación simple pero importante para un algoritmo de exponenciación mejorado, es que elevar al cuadrado un número es equivalente a multiplicar su exponente p por dos. Además, multiplicar dos números a^p y a^q es equivalente a $a^{(p+q)}$, es decir:

$$p = p_{b-1}2^{b-1} + ... + p_0 2^0$$

La idea principal de este algoritmo es considerar cada bit del exponente p y dividirlo entre dos hasta que llegue a cero, elevando al cuadrado el producto Q para cada bit. Además, si el bit actual es uno (es decir, si p es impar), entonces también multiplicamos Q por la base a :

$$Q = a^p \mod n$$

El número de llamadas recursivas y operaciones aritméticas es :

$$O\ (\log p)$$

Pseudo-Algoritmo:

A partir de ahora, nos referiremos a la exponenciación modular rápida por su traducción al inglés *FastExponentiation*

$Input: Integers\ a, p, n$

$Output: r = a^p mod\ n$

$if\ p = 0\ then$

  $return\ 1$

$if\ p\ is\ even\ then$

  $t \leftarrow FastExponentiation\ (a, p\ /\ 2, n)$

  $return\ t^2 mod\ n$

$t \leftarrow FastExponentiation\ (a, (p - 1)\ /\ 2, n)$

$return\ a(t^2 mod\ n)\ mod\ n$

Seguimiento numérico:

$$4^{10} mod\ 497$$

$$\begin{array}{ll} 4^{10} mod\ 497 & \rightarrow even \\ 4^{5}\ mod\ 497 & \rightarrow odd \\ 4^{2}\ mod\ 497 & \rightarrow even \\ 4^{1}\ mod\ 497 & \rightarrow odd \\ 4^{0}\ mod\ 497 = 1 & \end{array}$$

$$t^{2} mod\ n\ ,\ for\ even\ ;\ \ a(t^{2} mod\ n)\ mod\ n\ ,\ for\ odd$$

$$\begin{array}{lr} odd\ \rightarrow & 4 \times (1^{2} mod\ 497)\ mod\ 497 = 4 \\ even\ \rightarrow & 4^{2} mod\ 497 = 16 \\ odd\ \rightarrow & 4 \times (16^{2} mod\ 497)\ mod\ 497 = 30 \\ even\ \rightarrow & 30^{2} mod\ 497 = 403 \leftarrow result \end{array}$$

Implementación en C++:

```
ZZ FastExponentiation(ZZ a, ZZ p, ZZ n)
{
    ZZ t;
    if (p == 0)
        return ZZ(1);

    if ((p & 1) == 0)
    {
        t = FastExponentiation(a, p >> 1, n);
        return module(t * t, n);
    }
    t = FastExponentiation(a, (p - 1) >> 1, n);
    return module(a * module(t * t, n), n);
```

```
}
```

TEOREMA DE RESTO CHINO

Definición:

El teorema del resto chino es un teorema que da una solución única a las congruencias lineales simultáneas con módulos coprimos. En su forma básica, el teorema del resto chino determinará un número p que, cuando se divide por algunos divisores dados, deja residuos dados.

Supongamos que tenemos el siguiente sistema de congruencias simultáneas, donde $n, n_2, \ldots, n_k$ son enteros positivos y compromiso ; es decir que el maximo comun divisor de estos nk números agrupados de dos a dos es 1 .Entonces los enteros $a_1, a_2, \ldots, a_k$ existe un entero x que resuelve este sistema:

$$
\begin{aligned}
x &= a_1 \ (mod\ n_1) \\
x &= a_2 \ (mod\ n_2) \\
&\ldots \\
x &= a_k \ (mod\ n_k)
\end{aligned}
$$

Para comprobar su validez y encontrar el número de soluciones , se verifica si son coprimos a pares .

$$con\ mcd(a_i, a_j) = 1,\ 1 \leq i, j \leq k;\ entonces,\ si\ N = n_1, n_2, \ldots, n_k\ y\ N = N/n_i,$$

$$el\ sistema\ tiene\ solución\ única\ x = a_1 N_1 y_1 + a_2 N_2 y_2 + \ldots + a_k N_k y_k,\ módulo\ N$$

Pseudo-Algoritmo:

function TeoremaRestoChino(x , e , N):

1. p1,p2, .. pk deben ser primos entre sí
*2. P = p1 * p2*
3. P1 = P / p1
P2 = P / p2
...
Pk = P / pk
4.Para cada i existirá un qi
*Qi * Pi = 1 mod p1*

*5.Sea entonces x = (a1*P1*q1 + a2*P2*q2 + ... + ak*Pk*qk) mod P*
6.La solucion finalmente sera
*X = X + P*K*
return X

endfunction

Seguimiento Numérico:

Resolvemos el siguiente sistema

$$\begin{aligned} x &\equiv 1 \ (mod\ 3) \\ x &\equiv 2 \ (mod\ 5) \\ x &\equiv 3 \ (mod\ 7) \end{aligned}$$

$N = 3\ x\ 5\ x\ 7 = 105$, $N_1 = 35$, $N_2 = 21$, $N_3 = 15$. luego ,

$y_1 \equiv N_1^{-1} (mod\ 3) \Rightarrow y_1 \equiv 2\ (mod\ 3)$

$y_2 \equiv N_1^{-1} (mod\ 5) \Rightarrow y_1 \equiv 1\ (mod\ 5)$

$y_1 \equiv N_1^{-1} (mod\ 7) \Rightarrow y_1 \equiv 1\ (mod\ 7)$

Así $x = a_1 N_1 y_1 + a_2 N_2 y_2 + ... + a_k N_k y_k$

$x = 1\ x\ 35\ x\ 2 + 2\ x\ 21\ x\ 1 + ... + 3\ x\ 15\ x\ 1$

$x = 157 \equiv 52\ (mod\ M)$

Podemos decir , la solución única es $x \equiv 52\ (mod\ 105)$

Implementación en C++:

```
ZZ TRC(ZZ a, ZZ e, ZZ p, ZZ q)
{
    ZZ a1, a2, d1, d2, P, P1, P2, q1, q2, D;
    d1 = module(e, p - 1);
    d2 = module(e, q - 1);
    a1 = Left_to_Right_Binary(a, d1, p);
    a2 = Left_to_Right_Binary(a, d2, q);
    P = p * q;
    P1 = P / p;
    P2 = P / q;
    // q*P=1modp
```

```
    q1 = inverse(P1, p);
    q2 = inverse(P2, q);
    // D = a*P*q
    D = module(module(a1 * P1, P) * q1 + module(a2 * P2, P) * q2, P);
    return D;
}
```

Exponenciación por mínima cadena de sumas

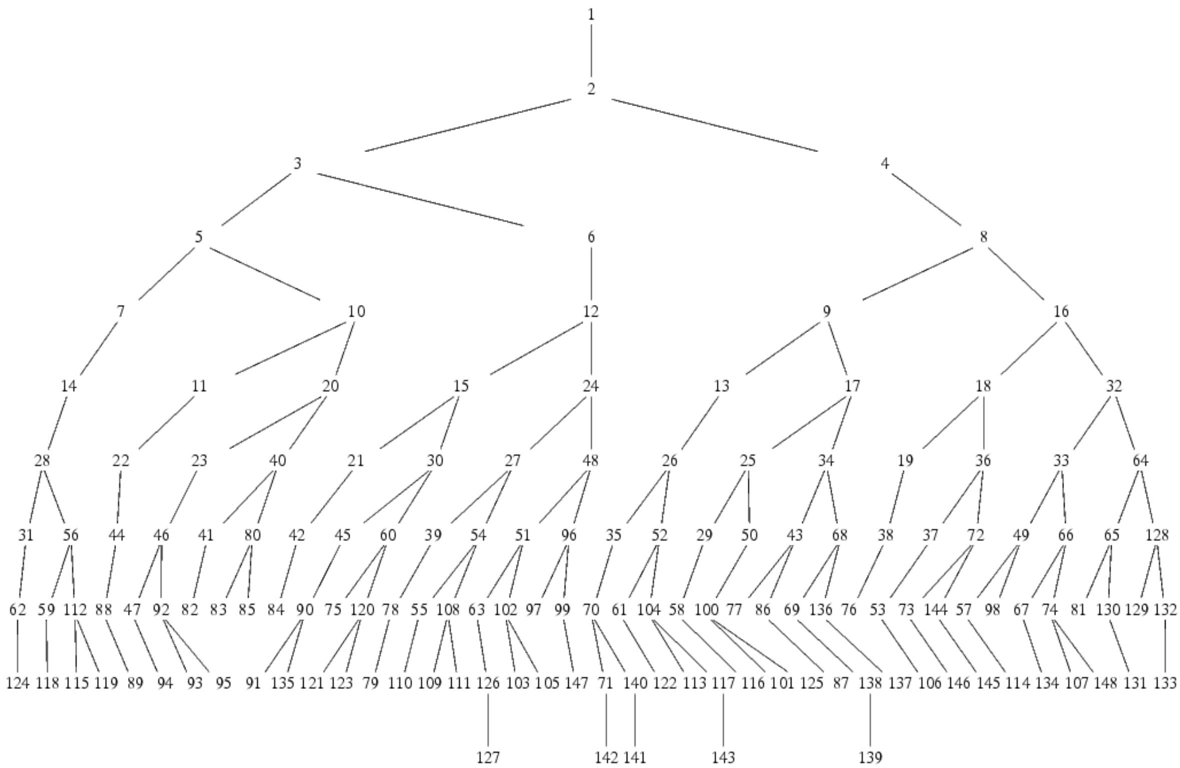
Definición:

Una cadena de suma para un entero positivo es un conjunto $1 = a_0 < a_1 <\ldots< a_r = n$ de enteros tales que para cada $i \geq 1$, $a_i = a_j + a_k$ para algún $k \leq j < i$ [6]

La manera más sencilla de encontrar una cadena de sumas es por medio de múltiplos de 2 y sumar la base al final si es necesario (lo que hace naive exponentiation). Existen cadenas de sumas aún más cortas que esa pero, sin embargo, no existe una única cadena óptima de sumas mínimas por número, ósea, tiene varias soluciones óptimas.

(Véase las imágenes del árbol de las cadenas más cortas de adición y el árbol de búsqueda para una mejor referencia)

A tree of Shortest Addition Chains for all n < 149



[7]

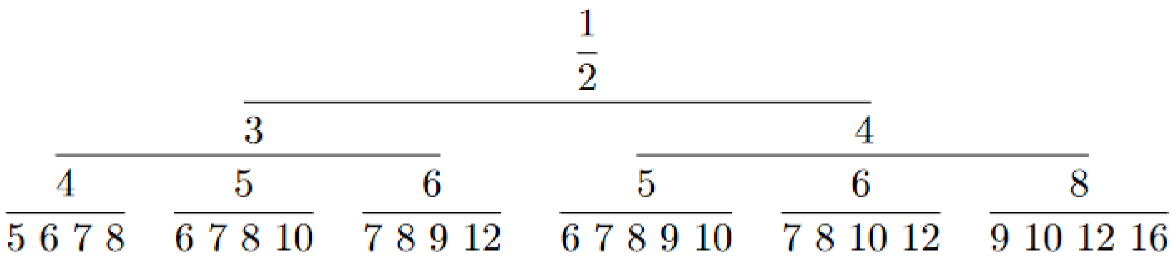
El problema fue originalmente planteado por Euler.[8]

Implementación

Términos:

Conjetura de Brauer: $\ell(2^n-1) <= n + \ell(n) - 1$
Aplicada a nuestra primera definición de cadena de sumas: k = i - 1
En términos generales, solo sumar al término más grande de la cadena. [9]

search es un árbol de búsqueda recursivo



[6]

Pseudocódigo

INPUT: base, exponent, maxDepth
OUTPUT: result
iterative depth-first search of Brauer sequence
FUNCTION search(chain, exponent, maxDepth)

1. chainSize > maxDepth
    1.1. return false
2. last = last element of the chain
3. LOOP i=0 , i<chainSize;
    3.1. sum = chain[chainSize -1 - i] + last
    3.2. if sum == exponent
        3.2.1. return true
    3.3. append sum chain
    3.4. if search(chain, exponent, maxDepth)
        3.4.1. return true
    3.5. delete the last element of chain
4. return false;

// increase depth until a solution is found
FUNCTION findChain(exponent)

1. map cache
2. if exponent está en cache
    2.1. return chain que encontró el exponente
3. else comienza búsqueda iterativa
    3.1. chain
    3.2. depth = 1
    3.3. WHILE TRUE

```
            3.3.1.    reset chain
            3.3.2.    if search(chain,exponent, depth)
                  3.3.2.1.    break;
            3.3.3.    depth++
      3.4.    cache[exponent] = chain
      3.5.    return chain
```

```
FUNCTION empower(base, exponent)
  1.    chain = findChain(exponent)
  2.    resultChain[chainSize]={base, base^2}
  3.    LOOP i = 2, i < chainSize, i++
      3.1.    sum = chain[i]
      3.2.    exp1_Index = i-1
      3.3.    exp2 = sum - chain[exp1_index]
      3.4.    LOOP j = 0, j < chainSize, j++
            3.4.1.    if exp2 == chain[j]
                  3.4.1.1.    exp2_index = j
                  3.4.1.2.    break
      3.5.    mult1 = resultChain[exp1_index]
      3.6.    mult2 = resultChain[exp2_index]
      3.7.    resultChain[i] = mult1 * mult2
  4.    return the last element of resultChain
```

[10]

Seguimiento Numérico:

```
->depth<-2
    last: 2
    LOOP    profundidad:0
    i:0     sum: 4  last: 2
    append          1, 2, 4,
    3 > 2
    pop             1, 2,
    i:1     sum: 3  last: 2
    append          1, 2, 3,
    3 > 2
    pop             1, 2,

->depth<-3
    last: 2
    LOOP    profundidad:0
    i:0     sum: 4  last: 2
    append          1, 2, 4,
    last: 4
    LOOP    profundidad:1
    i:0     sum: 8  last: 4
    append          1, 2, 4, 8,
    4 > 3
    pop             1, 2, 4,
    i:1     sum: 6  last: 4
    append          1, 2, 4, 6,
    4 > 3
    pop             1, 2, 4,
    i:2     sum: 5  last: 4
    append          1, 2, 4, 5,
    4 > 3
    pop             1, 2, 4,

    pop             1, 2,
    i:1     sum: 3  last: 2
    append          1, 2, 3,
    last: 3
    LOOP    profundidad:2
    i:0     sum: 6  last: 3
    append          1, 2, 3, 6,
    4 > 3
    pop             1, 2, 3,
    i:1     sum: 5  last: 3
    append          1, 2, 3, 5,
    4 > 3
    pop             1, 2, 3,
    i:2     sum: 4  last: 3
    append          1, 2, 3, 4,
    4 > 3
    pop             1, 2, 3,

    pop             1, 2,

->depth<-4
    last: 2
    LOOP    profundidad:0
    i:0     sum: 4  last: 2
    append          1, 2, 4,
    last: 4
    LOOP    profundidad:1
    i:0     sum: 8  last: 4
    append          1, 2, 4, 8,
    last: 8
    LOOP    profundidad:2
    i:0     sum: 16 last: 8
    append          1, 2, 4, 8, 16,
    5 > 4
    pop             1, 2, 4, 8,
    i:1     sum: 12 last: 8
    append          1, 2, 4, 8, 12,
    5 > 4
    pop             1, 2, 4, 8,
    i:2     sum: 10 last: 8
    append          1, 2, 4, 8, 10,
    5 > 4
    pop             1, 2, 4, 8,
    i:3     sum: 9  last: 8
    append          1, 2, 4, 8, 9,
    5 > 4
    pop             1, 2, 4, 8,

    pop             1, 2, 4,
    i:1     sum: 6  last: 4
    append          1, 2, 4, 6,
    last: 6
    LOOP    profundidad:3
    i:0     sum: 12 last: 6
    append          1, 2, 4, 6, 12,
    5 > 4
    pop             1, 2, 4, 6,
    i:1     sum: 10 last: 6
    append          1, 2, 4, 6, 10,
    5 > 4
    pop             1, 2, 4, 6,
    i:2     sum: 8  last: 6
    append          1, 2, 4, 6, 8,
    5 > 4
    pop             1, 2, 4, 6,
    i:3     sum: 7  last: 6
    append          1, 2, 4, 6, 7,
    5 > 4
    pop             1, 2, 4, 6,

    pop             1, 2, 4,
    i:2     sum: 5  last: 4
    append          1, 2, 4, 5,
    last: 5
    LOOP    profundidad:4
    i:0     sum: 10 last: 5
    append          1, 2, 4, 5, 10,
    5 > 4
    pop             1, 2, 4, 5,
```

```
i:1      sum: 9  last: 5
append          1, 2, 4, 5, 9,
5 > 4
pop             1, 2, 4, 5,
i:2      sum: 7  last: 5
append          1, 2, 4, 5, 7,
5 > 4
pop             1, 2, 4, 5,
i:3      sum: 6  last: 5
append          1, 2, 4, 5, 6,
5 > 4
pop             1, 2, 4, 5,

pop             1, 2, 4,

pop             1, 2,
i:1      sum: 3  last: 2
append          1, 2, 3,
last: 3
LOOP     profundidad:2
i:0      sum: 6  last: 3
append          1, 2, 3, 6,
last: 6
LOOP     profundidad:3
i:0      sum: 12 last: 6
append          1, 2, 3, 6, 12,
5 > 4
pop             1, 2, 3, 6,
i:1      sum: 9  last: 6
append          1, 2, 3, 6, 9,
5 > 4
pop             1, 2, 3, 6,
i:2      sum: 8  last: 6
append          1, 2, 3, 6, 8,
5 > 4
pop             1, 2, 3, 6,
i:3      sum: 7  last: 6
append          1, 2, 3, 6, 7,
```

```
5 > 4
pop             1, 2, 3, 6,

pop             1, 2, 3,
i:1      sum: 5  last: 3
append          1, 2, 3, 5,
last: 5
LOOP     profundidad:4
i:0      sum: 10 last: 5
append          1, 2, 3, 5, 10,
5 > 4
pop             1, 2, 3, 5,
i:1      sum: 8  last: 5
append          1, 2, 3, 5, 8,
5 > 4
pop             1, 2, 3, 5,
i:2      sum: 7  last: 5
append          1, 2, 3, 5, 7,
5 > 4
pop             1, 2, 3, 5,
i:3      sum: 6  last: 5
append          1, 2, 3, 5, 6,
5 > 4
pop             1, 2, 3, 5,

pop             1, 2, 3,
i:2      sum: 4  last: 3
append          1, 2, 3, 4,
last: 4
LOOP     profundidad:5
i:0      sum: 8  last: 4
append          1, 2, 3, 4, 8,
5 > 4
pop             1, 2, 3, 4,
i:1      sum: 7  last: 4
append          1, 2, 3, 4, 7,
5 > 4
pop             1, 2, 3, 4,
```

```
    i:2      sum: 6  last: 4
    append          1, 2, 3, 4, 6,
    5 > 4
    pop             1, 2, 3, 4,
    i:3      sum: 5  last: 4
    append          1, 2, 3, 4, 5,
    5 > 4
    pop             1, 2, 3, 4,

    pop             1, 2, 3,

    pop             1, 2,

->depth<-5
    last: 2
    LOOP     profundidad:0
    i:0      sum: 4  last: 2
    append          1, 2, 4,
    last: 4
    LOOP     profundidad:1
    i:0      sum: 8  last: 4
    append          1, 2, 4, 8,
    last: 8
    LOOP     profundidad:2
    i:0      sum: 16 last: 8
    append          1, 2, 4, 8, 16,
    last: 16
    LOOP     profundidad:3
    i:0      sum: 32 last: 16
    append          1, 2, 4, 8, 16, 32,
    6 > 5
    pop             1, 2, 4, 8, 16,
    i:1      sum: 24 last: 16
    append          1, 2, 4, 8, 16, 24,
    6 > 5
    pop             1, 2, 4, 8, 16,
    i:2      sum: 20 last: 16
    append          1, 2, 4, 8, 16, 20,
```

6 > 5
pop 1, 2, 4, 8, 16,
i:3 sum: 18 last: 16
append 1, 2, 4, 8, 16, 18,
6 > 5
pop 1, 2, 4, 8, 16,
i:4 sum: 17 last: 16
append 1, 2, 4, 8, 16, 17,
6 > 5
pop 1, 2, 4, 8, 16,

pop 1, 2, 4, 8,
i:1 sum: 12 last: 8
append 1, 2, 4, 8, 12,
last: 12
LOOP profundidad:4
i:0 sum: 24 last: 12
append 1, 2, 4, 8, 12, 24,
6 > 5
pop 1, 2, 4, 8, 12,
i:1 sum: 20 last: 12
append 1, 2, 4, 8, 12, 20,
6 > 5
pop 1, 2, 4, 8, 12,
i:2 sum: 16 last: 12
append 1, 2, 4, 8, 12, 16,
6 > 5
pop 1, 2, 4, 8, 12,
i:3 sum: 14 last: 12
append 1, 2, 4, 8, 12, 14,
6 > 5
pop 1, 2, 4, 8, 12,
i:4 sum: 13 last: 12
append 1, 2, 4, 8, 12, 13,
6 > 5
pop 1, 2, 4, 8, 12,

pop 1, 2, 4, 8,
i:2 sum: 10 last: 8
append 1, 2, 4, 8, 10,
last: 10
LOOP profundidad:5
i:0 sum: 20 last: 10
append 1, 2, 4, 8, 10, 20,
6 > 5
pop 1, 2, 4, 8, 10,
i:1 sum: 18 last: 10
append 1, 2, 4, 8, 10, 18,
6 > 5
pop 1, 2, 4, 8, 10,
i:2 sum: 14 last: 10
append 1, 2, 4, 8, 10, 14,
6 > 5
pop 1, 2, 4, 8, 10,
i:3 sum: 12 last: 10
append 1, 2, 4, 8, 10, 12,
6 > 5
pop 1, 2, 4, 8, 10,
i:4 sum: 11 last: 10
append 1, 2, 4, 8, 10, 11,
6 > 5
pop 1, 2, 4, 8, 10,

pop 1, 2, 4, 8,
i:3 sum: 9 last: 8
append 1, 2, 4, 8, 9,
last: 9
LOOP profundidad:6
i:0 sum: 18 last: 9
append 1, 2, 4, 8, 9, 18,
6 > 5
pop 1, 2, 4, 8, 9,
i:1 sum: 17 last: 9
append 1, 2, 4, 8, 9, 17,
6 > 5
pop 1, 2, 4, 8, 9,
i:2 sum: 13 last: 9
append 1, 2, 4, 8, 9, 13,
6 > 5
pop 1, 2, 4, 8, 9,
i:3 sum: 11 last: 9
append 1, 2, 4, 8, 9, 11,
6 > 5
pop 1, 2, 4, 8, 9,
i:4 sum: 10 last: 9
append 1, 2, 4, 8, 9, 10,
6 > 5
pop 1, 2, 4, 8, 9,

pop 1, 2, 4, 8,

pop 1, 2, 4,
i:1 sum: 6 last: 4
append 1, 2, 4, 6,
last: 6
LOOP profundidad:3
i:0 sum: 12 last: 6
append 1, 2, 4, 6, 12,
last: 12
LOOP profundidad:4
i:0 sum: 24 last: 12
append 1, 2, 4, 6, 12, 24,
6 > 5
pop 1, 2, 4, 6, 12,
i:1 sum: 18 last: 12
append 1, 2, 4, 6, 12, 18,
6 > 5
pop 1, 2, 4, 6, 12,
i:2 sum: 16 last: 12
append 1, 2, 4, 6, 12, 16,
6 > 5
pop 1, 2, 4, 6, 12,
i:3 sum: 14 last: 12
append 1, 2, 4, 6, 12, 14,

```
6 > 5
pop             1, 2, 4, 6, 12,
i:4     sum: 13 last: 12
append          1, 2, 4, 6, 12, 13,
6 > 5
pop             1, 2, 4, 6, 12,

pop             1, 2, 4, 6,
i:1     sum: 10 last: 6
append          1, 2, 4, 6, 10,
last: 10
LOOP    profundidad:5
i:0     sum: 20 last: 10
append          1, 2, 4, 6, 10, 20,
6 > 5
pop             1, 2, 4, 6, 10,
i:1     sum: 16 last: 10
append          1, 2, 4, 6, 10, 16,
6 > 5
pop             1, 2, 4, 6, 10,
i:2     sum: 14 last: 10
append          1, 2, 4, 6, 10, 14,
6 > 5
pop             1, 2, 4, 6, 10,
i:3     sum: 12 last: 10
append          1, 2, 4, 6, 10, 12,
6 > 5
pop             1, 2, 4, 6, 10,
i:4     sum: 11 last: 10
append          1, 2, 4, 6, 10, 11,
6 > 5
pop             1, 2, 4, 6, 10,

pop             1, 2, 4, 6,
i:2     sum: 8  last: 6
append          1, 2, 4, 6, 8,
last: 8
LOOP    profundidad:6
i:0     sum: 16 last: 8
append          1, 2, 4, 6, 8, 16,
6 > 5
pop             1, 2, 4, 6, 8,
i:1     sum: 14 last: 8
append          1, 2, 4, 6, 8, 14,
6 > 5
pop             1, 2, 4, 6, 8,
i:2     sum: 12 last: 8
append          1, 2, 4, 6, 8, 12,
6 > 5
pop             1, 2, 4, 6, 8,
i:3     sum: 10 last: 8
append          1, 2, 4, 6, 8, 10,
6 > 5
pop             1, 2, 4, 6, 8,
i:4     sum: 9  last: 8
append          1, 2, 4, 6, 8, 9,
6 > 5
pop             1, 2, 4, 6, 8,

pop             1, 2, 4, 6,
i:3     sum: 7  last: 6
append          1, 2, 4, 6, 7,
last: 7
LOOP    profundidad:7
i:0     sum: 14 last: 7
append          1, 2, 4, 6, 7, 14,
6 > 5
pop             1, 2, 4, 6, 7,
i:1     sum: 13 last: 7
append          1, 2, 4, 6, 7, 13,
6 > 5
pop             1, 2, 4, 6, 7,
i:2     sum: 11 last: 7
append          1, 2, 4, 6, 7, 11,
6 > 5
pop             1, 2, 4, 6, 7,
i:3     sum: 9  last: 7
append          1, 2, 4, 6, 7, 9,
6 > 5
pop             1, 2, 4, 6, 7,
i:4     sum: 8  last: 7
append          1, 2, 4, 6, 7, 8,
6 > 5
pop             1, 2, 4, 6, 7,

pop             1, 2, 4, 6,

pop             1, 2, 4,
i:2     sum: 5  last: 4
append          1, 2, 4, 5,
last: 5
LOOP    profundidad:4
i:0     sum: 10 last: 5
append          1, 2, 4, 5, 10,
last: 10
LOOP    profundidad:5
i:0     sum: 20 last: 10
append          1, 2, 4, 5, 10, 20,
6 > 5
pop             1, 2, 4, 5, 10,
i:1     sum: 15 last: 10
profundidad:4
profundidad:3
profundidad:0
```

la mínima cadena de sumas es: 1,2,4,5,10,15

2x = x*x ; 4x = 2x*2x; 5x = x*4x ; 10x = 5x * 5x;

15x = 10x*5

Implementation en C++

```
#include "power_module.hh"
#include <vector>
#include <map>
#include <NTL/ZZ.h>

using namespace std;
using namespace NTL;

// iterative depth-first search of Brauer sequence
bool PowerModule::search(Chain &chain, unsigned exponent, unsigned
maxDepth)
{
    // too deep ?
    if (chain.size() > maxDepth)
        return false;

    auto last = chain.back();
    for (size_t i = 0; i < chain.size(); i++)
    {
        //auto sum = chain[i] + last;
        auto sum = chain[chain.size() - 1 - i] + last; // try high
exponents first => about twice as fast
        if (sum == exponent)
            return true;

        chain.push_back(sum);
        if (search(chain, exponent, maxDepth))
            return true;

        chain.pop_back();
    }

    return false;
}
```

```cpp
// increase depth until a solution is found
Chain PowerModule::findChain(unsigned int exponent)
{
    // cached ? (needed for Hackerrank only)
    static std::map<unsigned int, Chain> cache;
    auto lookup = cache.find(exponent);
    if (lookup != cache.end())
        return lookup->second;
    // start iterative search
    Chain chain;
    unsigned int depth = 1;
    while (true)
    {
        // reset chain
        chain = {1};
        // a start search
        if (search(chain, exponent, depth))
            break;
        // failed, allow to go one step deeper
        depth++;
    }
    cache[exponent] = chain;
    return chain;
}

ZZ PowerModule::empower(ZZ base, unsigned exponent)
{
    auto chain = findChain(exponent);
    ZZ resultChain[chain.size()] = {base, base * base};
    for (unsigned i = 2; i < chain.size(); i++)
    {
        auto sum = chain[i];
        const unsigned exp1_index = i - 1;
        const unsigned exp2 = sum - chain[exp1_index];
        unsigned exp2_index;
        for (unsigned j = 0; j < chain.size(); j++)
```

```
            if (exp2 == chain[j])
            {
                exp2_index = j;
                break;
            }
        ZZ mult1 = resultChain[exp1_index];
        ZZ mult2 = resultChain[exp2_index];
        resultChain[i] = mult1 * mult2;
        // resultChain[i] = resultChain[i-1] *
resultChain[resultChain[i-1]]
    }
    return resultChain[chain.size() - 1];
}
```

[10]

Análisis de Algoritmos

Características del procesador y sistema operativo:

- AMD Ryzen 5 3400G with Radeon Vega Graphics (8 CPUs), ~3.7Ghz
- Windows 10 Pro 64 bits
- 16384MB RAM

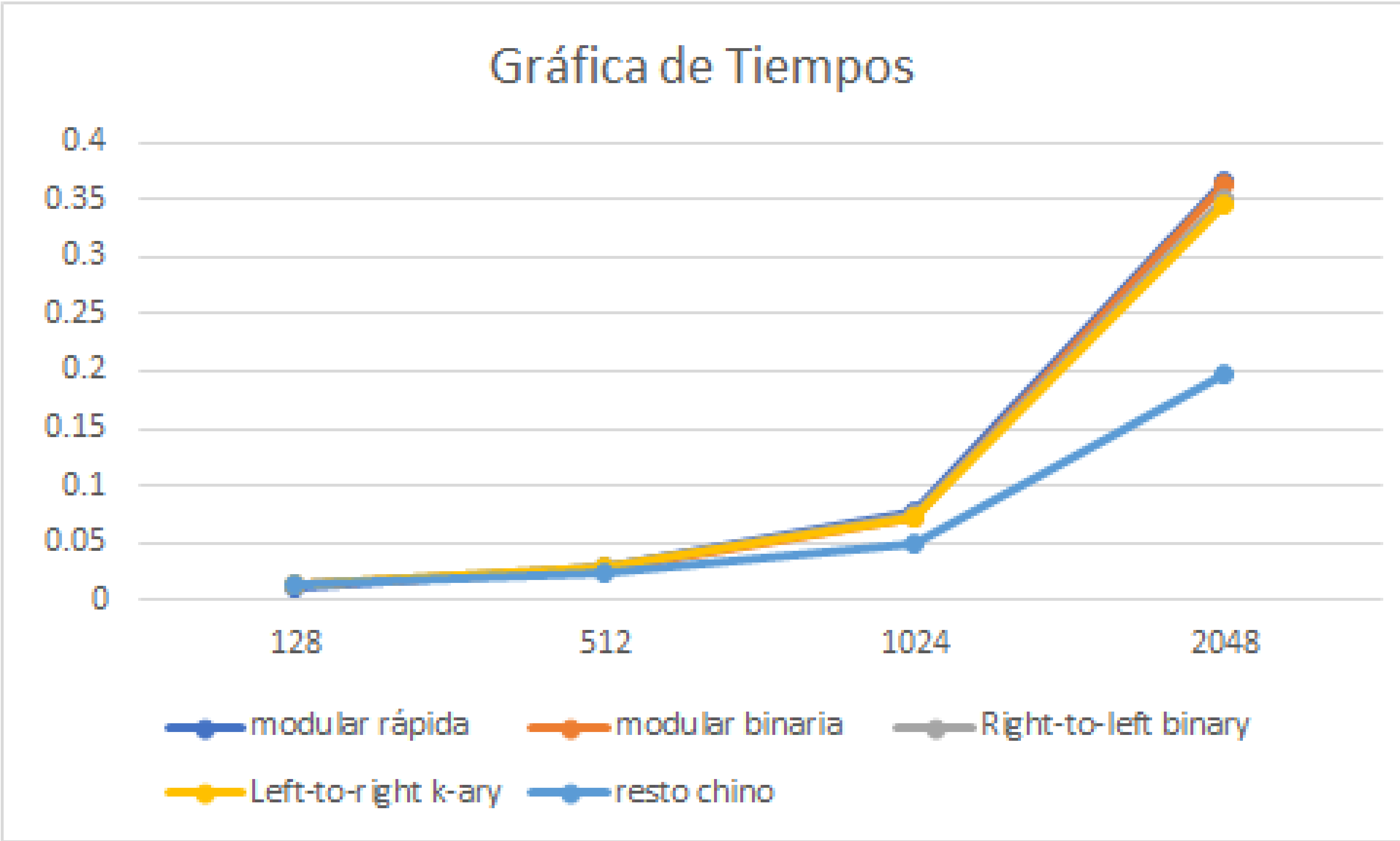
Tiempo de ejecución vs. Nro. de Bits

128 bits											
Intentos											Promedio
	1	2	3	4	5	6	7	8	9	10	
modular rápida	0,013	0,01	0,013	0,01	0,013	0,012	0,014	0,011	0,012	0,011	0,0119
modular binaria	0,013	0,012	0,015	0,012	0,013	0,013	0,013	0,011	0,013	0,01	0,0125
Right-to-left binary	0,013	0,015	0,013	0,011	0,014	0,011	0,016	0,012	0,012	0,013	0,013
Left-to-right k-ary	0,015	0,015	0,012	0,014	0,015	0,011	0,013	0,014	0,012	0,013	0,0134
resto chino	0,012	0,012	0,014	0,013	0,012	0,016	0,015	0,013	0,013	0,015	0,0135

512 bits											
Intentos											Promedio
	1	2	3	4	5	6	7	8	9	10	
modular rápida	0,03	0,028	0,029	0,028	0,029	0,028	0,025	0,026	0,03	0,032	0,0285
modular binaria	0,025	0,025	0,028	0,027	0,026	0,025	0,027	0,029	0,029	0,029	0,027
Right-to-left binary	0,031	0,031	0,027	0,026	0,028	0,027	0,029	0,028	0,028	0,032	0,0287
Left-to-right k-ary	0,027	0,028	0,027	0,028	0,028	0,03	0,027	0,028	0,028	0,028	0,0279
resto chino	0,022	0,023	0,02	0,022	0,025	0,025	0,021	0,022	0,024	0,023	0,0227

1024 bits											
Intentos											Promedio
	1	2	3	4	5	6	7	8	9	10	
modular rápida	0,083	0,09	0,076	0,082	0,072	0,072	0,075	0,07	0,081	0,074	0,0775
modular binaria	0,075	0,079	0,07	0,07	0,072	0,07	0,072	0,069	0,076	0,074	0,0727
Right-to-left binary	0,072	0,072	0,074	0,075	0,086	0,072	0,075	0,075	0,073	0,073	0,0747
Left-to-right k-ary	0,072	0,073	0,071	0,072	0,074	0,078	0,071	0,072	0,074	0,074	0,0731
resto chino	0,047	0,05	0,047	0,048	0,055	0,049	0,049	0,051	0,05	0,05	0,0496

2048 bits											
Intentos											Promedio
	1	2	3	4	5	6	7	8	9	10	
modular rápida	0,364	0,376	0,367	0,356	0,367	0,366	0,356	0,361	0,377	0,375	0,3665
modular binaria	0,371	0,365	0,377	0,352	0,379	0,371	0,358	0,36	0,35	0,351	0,3634
Right-to-left binary	0,34	0,348	0,355	0,371	0,329	0,353	0,371	0,34	0,346	0,346	0,3499
Left-to-right k-ary	0,349	0,34	0,355	0,347	0,335	0,346	0,347	0,34	0,343	0,352	0,3454
resto chino	0,229	0,214	0,193	0,185	0,194	0,195	0,193	0,184	0,199	0,193	0,1979


Gráfica de Tiempos
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
128
512
1024
2048
modular rápida
modular binaria
Right-to-left binary
Left-to-right k-ary
resto chino

Conclusión:

Se concluye que el algoritmo con mejor desempeño es el teorema del resto chino , debido a que en vez de usar cálculos mod N , hacemos módulo p y q , cuyos tamaños son la mitad de bits de N , los cálculos se realizan una sola vez con la posibilidad de hacerlos en paralelo . Aunque el tamaño de la clave privada es de orden de tamaño n , los posteriores tamaños de los exponentes dp y dq son la mitad de bits , además de poder utilizar algoritmos eficientes para encontrar su inversa , basándose en el teorema de Fermat .

Aun teniendo en cuenta las claras ventajas , se observa que a menor cantidad de bits , el algoritmo se mantiene dentro del promedio de tiempo con sus pares evaluados ; por tal motivo dados las evaluaciones de tiempo frente a la cantidad de bits , se recomienda para números grandes especialmente con 1024 y 2048 bits donde es 2 veces más rápido que sus pares .

Bibliografía:

[03] Handbook of Applied Cryptography, Menezes, Oorschot, Vanstone. CRC Press, New York, fifth edition (2001). chap14.pdf (uwaterloo.ca)

[04] Chapter 10. Number theory and Cryptography.
[PDF] Chapter. Number Theory and Cryptography. Contents - Free Download PDF (silo.tips)

[05] Fast Exponentiation Article Microsoft Word - fastexp.doc (uic.edu)

[6] Thurber, E. G. (1999). Efficient Generation of Minimal Length Addition Chains. *SIAM Journal on Computing*, *28*(4), 1247–1263. https://doi.org/10.1137/s0097539795295663

[7] Clift, N. (n.d.). *Addition Chains*. Http://Additionchains.Com/. Retrieved June 11, 2021, from http://additionchains.com/

[8] Hughes, C. (2011, June 11). *Problem 122 - Project Euler*. Euler. https://projecteuler.net/problem=122

[9] Clift, N. (n.d.). *Addition Chains*. Http://Additionchains.Com/. Retrieved June 11, 2021, from http://additionchains.com/

[10]Brumme, S. (2017, May 15). *My C++ solution for Project Euler 122: Efficient exponentiation*. Http://Euler.Stephan-Brumme.Com/122/. http://euler.stephan-brumme.com/122/