CIENCIA DE LA COMPUTACIÓN

TRABAJO DE INVESTIGACIÓN E IMPLEMENTACIÓN DEL ALGORITMO DE EXPONENCIACIÓN MODULAR

ÁLGEBRA ABSTRACTA

DANIELA CHAVEZ AGUILAR

GIULIA NAVAL FERNANDEZ

PAOLO DELGADO VIDAL

RODRIGO TORRES SOTOMAYOR

PABLO CARAZAS BARRIOS

AÑO 2021

RESUMEN

En el presente trabajo se muestra un análisis y comparación de diversas versiones del algoritmo de exponenciación modular, aplicando distintas bases matemáticas y algunos teoremas como el Pequeño Teorema de Fermat y el de Euler, como se apreciará más adelante. Entre los algoritmos que se implementaron durante el desarrollo de la investigación están:

- Algoritmo de exponenciación rápida
- Algoritmo de Fuerza Bruta
- Exponenciación modular binaria right-to-left
- Exponenciación modular binaria left-to-right
- Exponenciación m-aria left-to-right
- Aplicación de Euler-Fermat
- Teorema del Resto Chino

Para hacer la comparación de estos algoritmos se utilizó como criterio el tiempo de ejecución, el número de bucles efectuados y el número de variables utilizadas con entradas con diferente número de bits. Al final de esta evaluación, el algoritmo con el mejor desempeño fue el algoritmo de exponenciación modular binario de derecha a izquierda.

1. INTRODUCCIÓN

En el presente trabajo se resolverá la siguiente pregunta: ¿Cuál es el algoritmo de exponenciación modular más eficiente? Por ende, los objetivos serán: analizar y comparar diferentes algoritmos de exponenciación rápida modular.

CONTENIDO TEÓRICO

ALGORITMOS DE EXPONENCIACIÓN

A. Algoritmo de exponenciación rápida

Definición:

Algoritmo de exponenciación modular con la característica de que sólo operará empleando exponentes múltiplos de 2. Permite reducir la cantidad de multiplicaciones a las necesarias.

Base Matemática:

El algoritmo de exponenciación rápida solo hará uso del teorema de la división (módulo), con el cual determinará en cuando es necesario realizar una operación (cuando tengamos un exponente impar), así como se aplicará a el resultado de cada multiplicación:

$$a^{e} mod\, m \quad e = e/2 \quad a = a * a \, mod\, m$$

$$mod \;\Rightarrow\; r = a - q * b$$

Donde nosotros denotaremos la exponenciación como: $(a * exp)\ mod\ m$repetido cada vez que el módulo entre e y 2 sea igual a 1.

Seguimiento:

a=124 e=12 m=214 resultado=34

a	e	m	División(e,2)	exp
124	12	214	0	1
182	6	214	0	1
168	3	214	1	1
190	1	214	1	168
148	0	214	1	34

Pseudo-Algoritmo:

Entrada: 3 enteros positivos a,e y m

Salida: exp modular

- exp←1
- a ← division(a,m)
- **Mientras** e sea mayor a 0:
    - **Si** division(e,2) es igual a 1:
        - exp ← division((exp*a),m)
    - e←e/2
    - a←division((a*a),m)
- **return** exp

Código C++:

```
ZZ exponenciacion_rapida(ZZ num, ZZ e, ZZ mod){

        ZZ exp; exp=1;

   num = division(num,mod);

        while(e>0){

                if(division(e,conv<ZZ>(2))==conv<ZZ>(1)){

                        exp = division((exp*num),mod);

                }

                e=(e)/2;

                num =division((num*num),mod);
```

```
        }

        return exp;

    }
```

B. Algoritmo en fuerza bruta(*Naive Exponentiation*)

_____Definición:

Exponenciación modular directa, la cual realiza todas las multiplicaciones acorde al número del exponente, aplicando módulo en todas y cada una de ellas.

Base Matemática:

El algoritmo de exponenciación en fuerza bruta o naive exponentiation solo hará uso del teorema de la división (módulo) al momento de operar, con el cual realizará multiplicaciones consecutivas sobre sí mismo en base a un exponente y tendrá esta forma:

$$a^{e} mod\, m \qquad a = q * b + r \qquad mod = r = a - q * b$$

Donde nosotros denotaremos la exponenciación como: $(a * exp)\, mod\, m$ repetido “e” veces.

Seguimiento:

a=124 e=12 m=214 resultado=34

i	exp	a	e	m
1	124	124	12	214
2	182	124	12	214
3	98	124	12	214
4	168	124	12	214
5	74	124	12	214
6	188	124	12	214
7	200	124	12	214
8	190	124	12	214
9	20	124	12	214
10	126	124	12	214
11	2	124	12	214
12	34	124	12	214

Pseudo-Algoritmo:

Entrada: 3 enteros positivos a,e y m

Salida: exp modular

- exp←1
- i←1
- **Para** i desde 1 hasta e:
    - exp←division((a*exp),m)
- **return** exp

Código C++:

```
#include<iostream>

#include <NTL/ZZ.h>

#include<cstdlib>

using namespace std;

using namespace NTL;

ZZ naive_exponentiation(ZZ num, ZZ e, ZZ mod){
```

```
    ZZ exp,i; exp=1;
    for(i=1;i<=e;i++){
       exp = division((num*exp),mod);
    }

    return exp;
}
```

C. Exponenciación modular binaria right-to-left

Definición y fundamento

Este algoritmo se fundamenta en el hecho de que si el exponente e es par, se puede calcular $g^e \bmod n$ como $(g^{e/2} * g^{e/2}) \bmod n$, 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.

Pseudo-algoritmo

ENTRADA: un elemento $g \in G$, un entero $e \geq 1$ y un módulo n

SALIDA: $g^e \bmod n$

$A \leftarrow 1,\ S \leftarrow g$

Mientras $e \neq 0$:

a. Si *e* es impar: $A \leftarrow A \cdot S \bmod n$
b. $e \leftarrow \lfloor e/2 \rfloor$
c. Si $e \neq 0$: $S \leftarrow S \cdot S \bmod n$

Retorna A

Seguimiento

Entrada: g=25, e=5, n=28

A	S	e
1	25	5
25	9	2
25	25	1
9	25	0

Salida: 9

Código

```
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;
}
```

D. Exponenciación modular binaria left-to-right

Definición y fundamento

Este algoritmo se fundamenta en el hecho de que si el exponente e es par, se puede calcular $g^e \ mod \ n$ como $(g^{e/2} * g^{e/2}) \ mod \ n$, 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 esta versión, el exponente e se recibe como una cadena con la representación binaria del número.

Pseudo-algoritmo

ENTRADA: un elemento $g \in G$, un entero $e = \left(e_t e_{t-1} \ldots e_1 e_0 \right)_2$ y un módulo n

SALIDA: $g^e \ mod \ n$

1. $A \leftarrow 1$
2. Por cada i desde t hasta 0:
   a. $A \leftarrow A \cdot A \ mod \ n$
   b. Si $e_i = 1$: $A \leftarrow A \cdot g \ mod \ n$
3. Retorna A

Seguimiento

Entrada: g=14, e=6(0110), n=40

A	e_i
1	0
14	1
24	1
16	0

Salida: 16

Código

```
ZZ left2right_binary_modexp(ZZ g, string e, ZZ n){
  ZZ A=conv<ZZ>(1);
  for (int i=t; i>0; i--){
        A=divi(A*A,n);
        if (e[t-i]=='1')
        A=divi(A*g,n);
  }
  return A;
}
```

E. Exponenciación m-aria left-to-right

Definición y fundamento

Este algoritmo es una generalización del algoritmo de exponenciación modular binario left-to-right, con la mejora de que procesa más de un bit del exponente a la vez. Además es más eficiente al realizar cálculos durante la precomputación, que se utilizan varias veces más adelante en el algoritmo.

Pseudo-algoritmo

ENTRADA: un elemento $g \in G$, un entero $e = \left(e_t e_{t-1} \ldots e_1 e_0 \right)_b$, donde $b = 2^k$ para algún $k \geq 1$ y un módulo n

SALIDA: $g^e \ mod \ n$

1. Precomputación
    a. $g_0 = 1$
    b. Por cada i desde 1 hasta $2^k - 1$: $g_i = g_{i-1} \cdot g$
2. $A = 1$
3. Por cada i desde t hasta 0:
    a. $A = A^{2^k} \ mod \ n$
    b. $A = A \cdot g_{e_i} \ mod \ n$
4. Retorna A

Seguimiento

Entrada: g=685, e=15(1111), n=40

A	e_i
1	685
5	685
25	685
5	1

Salida: 5

Código

```
ZZ left2right_4ary_modexp(ZZ g, string e, ZZ n){
  ZZ G[16]; G[0]=ZZ(1);
  for (int i=1; i<=15; i++){
        G[i]=G[i-1]*g;
  }
  ZZ A=conv<ZZ>(1);
  for (int i=t; i>0; i--){
        A=left2right_binary_modexp(A, Num2Bits(conv<ZZ>(16),t),n);
        A=divi(A*G[e[t-i]-'0'],n);
  }
```

```
  return A;
}
```

F. Exponenciación modular aplicando Euler-Fermat

Definición

Este algoritmo es una mejora al algoritmo binario de exponenciación modular, aunque podría funcionar con cualquier otro algoritmo.

Base matemática

El algoritmo se apoya en el Pequeño Teorema de Fermat que nos dice que si p es un número primo y a es un número entero positivo, coprimo con p, entonces:

$$a^{p-1} \equiv 1 \ mod \ p \ \text{ o } \ a^p \equiv a \ mod \ p$$

Y en la generalización de este teorema, el Teorema de Euler, que nos dice que si a, n son 2 números coprimos entonces:

$$a^{\varphi(n)} \equiv 1 \ mod \ n$$

Pseudo-algoritmo

ENTRADA: un elemento $g \in G$, un entero $e \geq 1$ y un módulo n

SALIDA: $g^e \ mod \ n$

Si $mcd(b, m) = 1$:

Si n es primo:

Retorna $g^{e \ mod \ (m-1)} \ mod \ n$

Retorna $g^{e \ mod \ \varphi(n)} \ mod \ n$

Retorna $g^e \ mod \ n$

Código

```
bool Fermat_primo(ZZ n, int k){
  for (int i=0; i<k; i++){
        ZZ a=divi(conv<ZZ>(rand()),n)+2;
        if (right2left_binary_modexp(a, n-1, n)!=1)
        return false;
```

```
  }
  return true;
}
ZZ Euler_phi(ZZ n){
        ZZ a,f,e;
a  =  n; f = 3; e = n;
bool tomar_primo;
  if (divi(n,conv<ZZ>(2))==0)
        e>>=1;
  while (f <= a){
        if (Fermat_primo(f,3)){
        tomar_primo = true;
        while (divi(a,f)==0){
        a = a / f;
        if (tomar_primo){
        e = e * (f -1) / f;
        tomar_primo = false;
        }
        }
        }
        f = f + 2;
  }
return e;
}
ZZ expmod_fermat_euler(ZZ b, ZZ e, ZZ m){
  if (Euclides(b,m)==1){
        if (Fermat_primo(m, 3)){
                return right2left_binary_modexp(b,divi(e,m-1),m);}
        return right2left_binary_modexp(b,divi(e,Euler_phi(m)),m);
```

```
  }
  return right2left_binary_modexp(b,e,m);
}
```

OTROS ALGORITMOS:

A. Teorema del resto chino

Definición:
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:

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

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

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

$$N = md_1 md_2 \dots md_k \; N_i = \frac{N}{md_i}$$

$$Existen\ dos\ enteros\ r_i\ y\ s_i\ que\ permiten: r_i md_i + s_i N_i = 1$$

$$s_i N_i \equiv 1\ mod\ md_i \; s_i N_i \equiv 0\ mod\ md_j$$

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

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

Observaciones:

Debido a que las características y condiciones propias del teorema del resto chino, este algoritmo solo nos brindaría una exponenciación modular correcta bajo requisitos específicos (tales como los que nos brinda el rsa al momento de operar) y por ende resultaría ineficiente e incluso contraproducente utilizar dicho algoritmo para una exponenciación modular con enteros que no cumplan con las condiciones dadas.

Código C++:

```
#include <NTL/ZZ.h>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <conio.h>
#include <cmath>
using namespace std;
using namespace NTL;

const int tam = 3;
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;
}

vector<ZZ> factores_phi(ZZ n){
ZZ a  =  n, f = conv<ZZ>(3);
vector <ZZ> factors;
bool tomar_primo;
  if (divi(n,ZZ(2))==0){
          factors.push_back(ZZ(2));//un factor primo es 2
  }
  while (f <= a){
          if (Fermat_primo(f,3)){
          while (divi(a,f)==0){
          a = a / f;
          factors.push_back(f);
          }
          }
          f = f + 2;
  }
return factors;
}


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. Exponenciación (Iterativo)

Definición:

Algoritmo sencillo que encuentra iterativamente la exponenciación de un número a módulo n.

Base matemática:

$$a^n = a \cdot a \ldots \cdot a$$
$$n \text{ veces}$$
$$\left[(a \bmod n) \cdot (b \bmod n) \right] \bmod n \equiv (a \cdot b) \bmod n$$

Seguimiento:

a=2 p=4 n=10

i	r	a	p	n
1	2	2	4	10
2	4	2	4	10
3	8	2	4	10
4	6	2	4	10

Pseudo-Algoritmo:

ENTRADA: Enteros $a,\ p,\ n$

SALIDA: $r = a^p \bmod n$

$r = 1$

Para un $i = 1$hacia un p:

$r = (r.a) \bmod n$

Retorna r

Código C++:

```
ZZ Exponenciacion(ZZ a,ZZ p,ZZ n){
```

```
        ZZ r;// salida: r= modulo(pow(a, p) ,n);
        r=1;
        for (int i=1 ; i <= p;i++){
                r=modulo((r*a),n);
        }
        return r;
}
```

C. Exponenciación (recursivo)

Definición:

Algoritmo sencillo que encuentra recursivamente la exponenciación de un número a módulo n.

Base matemática:

$$a^n = a \cdot a \ldots \cdot a$$
$$n \ veces$$
$$\left[(a \ mod \ n) \cdot (b \ mod \ n) \right] mod \ n \equiv (a \cdot b) \ mod \ n$$

Seguimiento:

a=5 p=10 n=17

a	p	n
5	10	17
5	5	17
5	2	17
5	1	17

salida: 9

Pseudo-Algoritmo:

ENTRADA: Enteros $a, \ p, \ n$

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

Si $p = 0$:

Retorna 1

Si $p\ es\ par$:

$t = exponenciacion(a,\ p/2,\ n)$

Retorna $t^2\ mod\ n$

$t = exponenciacion\ (a,\ (p - 1)/2,\ n)$

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

Código C++:

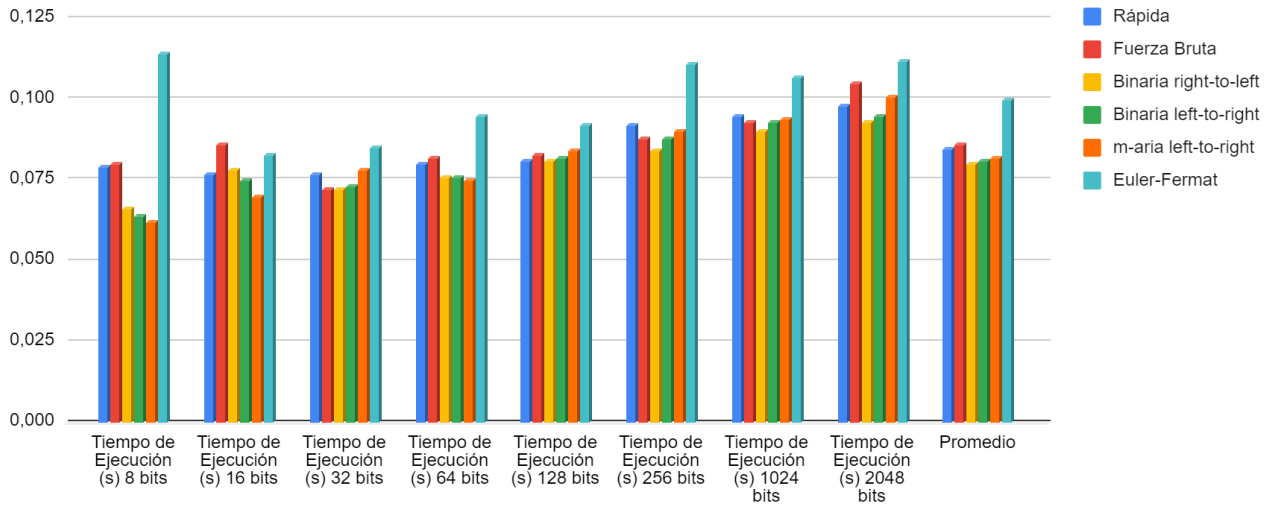
```
ZZ Exponenciacion2 (ZZ a,ZZ p,ZZ n){
	ZZ r; //salida: r= modulo(pow(a, p) ,n)
	ZZ t;
	if (p==0){
		return ZZ(1);
	}
	if (modulo(p,2)==0){
		t= Exponenciacion2 (a,p/2,n);
		return modulo(t*t,n);
	}
	t= Exponenciacion2 (a,(p-1)/2,n);
	return modulo(a*(modulo(t*t,n)),n);
}
```

ANÁLISIS DE LOS ALGORITMOS

Para los algoritmos utilizamos: Codeblocks usando la librería NTL para probar con los distintos números de bits.

Sistema Operativo: Windows 10

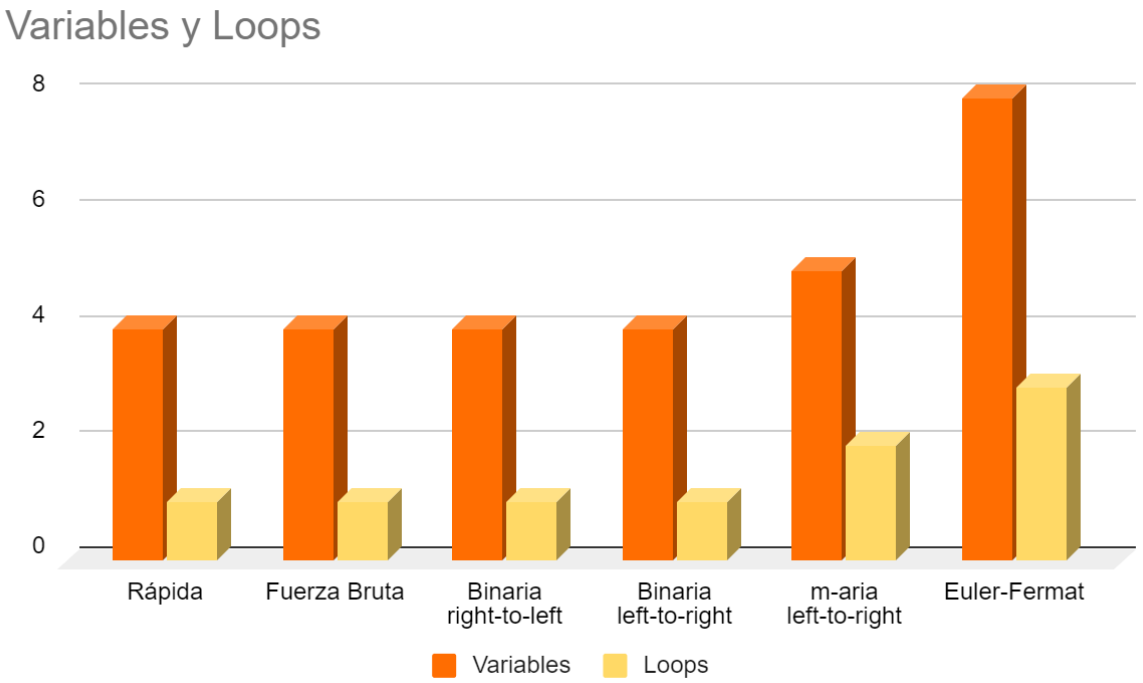
	Rápida	Fuerza Bruta	Binaria right-to-left	Binaria left-to-right	m-aria left-to-right	Euler-Fermat
Tiempo de Ejecución (s) 8 bits	0,079	0,080	0,066	0,064	0,062	0,114
Tiempo de Ejecución (s) 16 bits	0,077	0,086	0,078	0,075	0,07	0,083
Tiempo de Ejecución (s) 32 bits	0,077	0,072	0,072	0,073	0,078	0,085
Tiempo de Ejecución (s) 64 bits	0,080	0,082	0,076	0,076	0,075	0,095
Tiempo de Ejecución (s) 128 bits	0,081	0,083	0,081	0,082	0,084	0,092
Tiempo de Ejecución (s) 256 bits	0,092	0,088	0,084	0,088	0,09	0,111
Tiempo de Ejecución (s) 1024 bits	0,095	0,093	0,090	0,093	0,094	0,107
Tiempo de Ejecución (s) 2048 bits	0,098	0,105	0,093	0,095	0,101	0,112
Promedio	0,085	0,086	0,080	0,081	0,082	0,100



CANTIDAD DE VARIABLES Y LOOPS

	Rápida	Fuerza	Binaria	Binaria	m-aria	Euler-Fer

		Bruta	right-to-left	left-to-right	left-to-right	mat
Variables	4	4	4	4	5	8
Loops	1	1	1	1	2	3



CONCLUSIONES GENERALES

Tras finalizar el análisis de algoritmos y la comparación de los mismos, fuimos capaces de apreciar cómo los distintos enfoques matemáticos que se utilizaron hacen más o menos eficientes diversos procesos con un mismo fin: efectuar la

exponenciación modular de un número. Y con ello, pudimos verificar que el algoritmo más eficiente es el de exponenciación modular binario de derecha a izquierda.

REFERENCIAS

[01] Handbook of Applied Cryptography, Menezes, Oorschot, Vanstone. CRC Press, New York, fifth edition (2001). http://cacr.uwaterloo.ca/hac/about/chap14.pdf

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

[03] Introducción a la Teoría de Números. Ejemplos y algoritmos. Walter Mora. Capítulo 4 https://repositoriotec.tec.ac.cr/bitstream/handle/2238/6299/introducci%C3%B3nteor%C3%ADa-n%C3%BAmeros.pdf?sequence=1&isAllowed=y

[04] Teorema de Euler. https://es.wikipedia.org/wiki/Teorema_de_Euler

[05] TEOREMA DE EULER-FERMAT. https://www.youtube.com/watch?v=rH7h59xZMus

Enlace GitHub: https://github.com/Ni81194/Daniela-Alejandra-Ch-vez-Aguilar/tree/main