Exponenciación modular

Autores:

- Brigham Jeffrey Caceres Gutierrez
- Galo de Paula Jimenez Araujo
- Fernando Jair Peralta Bustamante

Resumen:

Esta es una comparación sencilla de los algoritmos de exponenciación modular donde el más eficiente será usado para el algoritmo de RSA que se presentará al final del curso de Álgebra abstracta.

Introducción:

La exponenciación modular es un tipo de exponenciación realizada sobre un módulo. Es particularmente útil en ciencias de la computación, especialmente en el campo de la criptografía.

En este ensayo los investigadores vamos a presentar un análisis de los siguientes algoritmos para determinar la exponenciación modular:

1. Exponenciación modular rápida
2. Exponenciación modular binaria
3. Teorema del resto chino.
4. Right-to-left binary exponentiation.

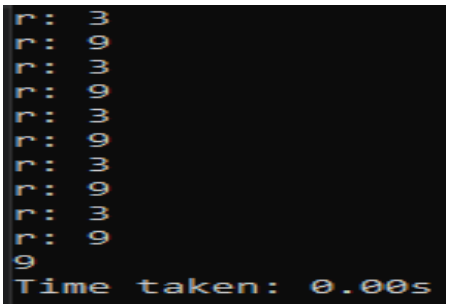
Exponenciación modular rápida

En la exponenciación modular rápida, también conocida a veces como la exponenciación modular bruta se busca hacer el proceso de módulo a varias iteraciones de la exponenciación del número en cuestión para siempre estar en Z_n. Es decir, si tenemos 3^{10} mod 12, se hace la multiplicación de 3^2 10 veces, y cada vez se resuelve el módulo de la operación.

Código:

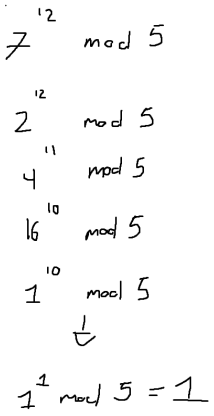
```
ZZ Expo_Bruta(ZZ a, ZZ b, ZZ c) {
        ZZ r(1);
        ZZ p(b);
        ZZ n(c);
```

```
        for (ZZ i(0); i < p; i++) {
                r = (r * a) % n;
                cout << "r: " << r << endl;
        }
        return r;
}
```



Las limitaciones de este algoritmo se dan en su escalabilidad dado que al superar los 256 bits el programa se cuelga.

Ejemplo:



Exponenciación modular binaria

En la exponenciación modular binaria se busca hacer varias modulaciones con diferentes valores de p. Con este método nos basamos que la exponencial de a y p también se puede interpretar desde un nivel de índices, y basado en este podemos decir que $(a^p)^2$ es lo mismo que decir a^{p*2}. Basándonos en esto podremos reducir drásticamente los valores de p, que son los valores más problemáticos en este tipo de problemas dado que determinan el número de iteraciones que se tendrá que hacer.

Es por esto que se puede evaluar una exponenciación con un método llamado exponenciación al cuadrado repetitivo. Básicamente funciona trabajando con a la par el índice de exponenciación y el producto. Si dividimos el exponente entre 2, entonces el producto debe ser el resultado de la división entre dos multiplicado por sí mismo. Dado que 2^8 es lo mismo que 2^4 x 2^4. Esto busca separar el producto hasta que p = 1, y luego multiplicarlos de manera ordenada hasta tener la respuesta final.

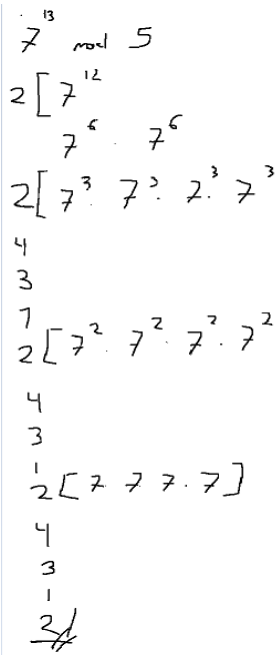
Código:

```
ZZ Expo_Bino(ZZ a, ZZ b, ZZ c){
  ZZ p(b);
        ZZ n(c);
        if (p == ZZ(0))
        {
        return ZZ(1);
        }
```

```
        else if (modular_galo(p, ZZ(2)) == 0) { // esto se debería hacer con bitwise
        ZZ t(Expo_Bino(a, (p / 2), n));
        return (modular_galo((t * t), n));
        }
        else {
        ZZ t(Expo_Bino(a, (p - 1), n));
        return modular_galo((a*(modular_galo((t * t), n))),n);
        }
}
```

Al ser un programa recursivo tiene un costo computacional más alto que uno iterativo.

Ejemplo:



Teorema del resto chino

Se basa en un sistema de congruencia modular donde se quiere hallar el valor de “X”:

$$X \equiv a(1)(\bmod n(1))$$

$$X \equiv a(2)(\bmod n(2))$$

…

$$X \equiv a(n)(\bmod n(n))$$

Se conoce el resto de la división de x entre un grupo de números desde 0 <= X < N , N = n(1)n(2)..n(n). Si los números nos son coprimos por pares entonces la ecuación tiene exactamente una solución,es por esto que se puede aplicar perfectamente al RSA en la parte de descifrado, se conoce el resto de la división y además lo conforman primos.

Código:

```cpp
#include <iostream>
#include <NTL/ZZ.h>
#include <vector>
#include "math.cpp"

using namespace std;
using namespace std;

ZZ restoChinoGauss(vector<ZZ> n, ZZ N, vector<ZZ> a)
{

   ZZ result(0);
   for(int i = 0; i < n.size(); i++)
   {
       ZZ bi = N / n[i];
       result += a[i] * bi * modulo(euclides_extendido(bi, n[i])[0],
N);
   }

   return  modulo(result, N);
}
```

Ejemplo:

Right to left binary exponentiation

El algoritmo consiste en un elemento *g* y un entero *e* mayor o igual a uno que vendría a ser el número exponente.

$$S \rightarrow g \rightarrow g^2 \rightarrow g^4 \rightarrow g^8 \rightarrow g^{16} \rightarrow g^{32}$$

e se divide entre dos y, por cada iteración, mientras *e* sea mayor que cero, en S se incrementa la exponenciación binaria.

$$e \geq 0 \Rightarrow e/2, \text{ indiferente si } e \text{ es impar. (binario)}$$

Se crea otra variable A donde se dará el resultado final (g^e).

$$A = 1, \text{ si } e \bmod 2 = 1 \Rightarrow A = A*S = 1*g$$

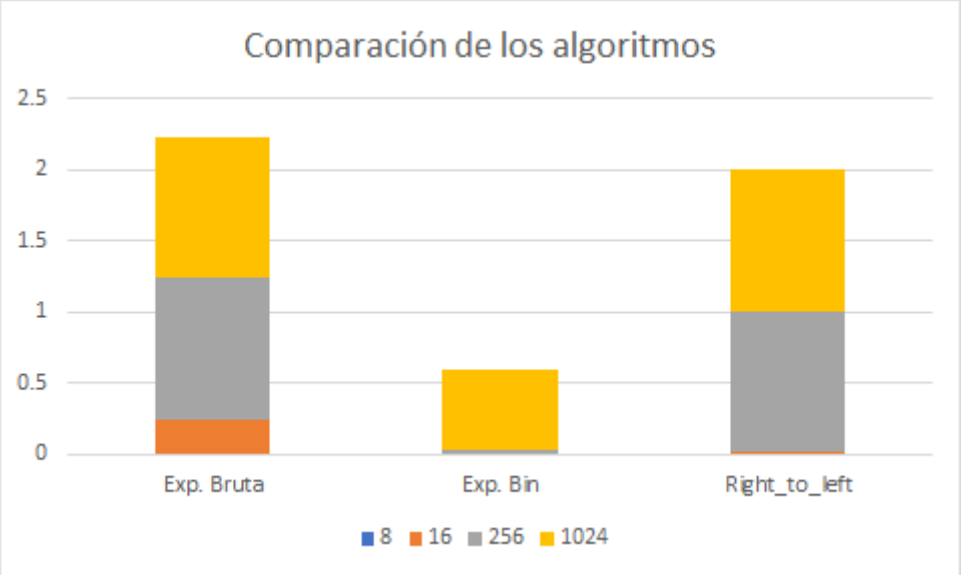
Código:

```
ZZ Right_to_left_binary_exponentation(ZZ g, ZZ e){

    ZZ A = to_ZZ(1);
    ZZ S = g;

    while (e != 0) {
        if (bit(e,0)==1)
        {
            A = A * S;
        }
        e = e >> 1;
        if (e!=0)
        {
            S = S * S;
        }
    }

    return A;
}
```

Ejemplo:

A	1	g	g^3	g^3	g^{11}	g^{27}	g^{27}	g^{27}	g^{27}	g^{283}
e	283	141	70	35	17	8	4	2	1	0
S	g	g^2	g^4	g^8	g^{16}	g^{32}	g^{64}	g^{128}	g^{256}	–

Comparación:



La comparación se llevó a cabo en estos tres algoritmos dado que el teorema del resto chino requiere de una mayor cantidad de factores lo cual hace desfavorable la implementación en la comparación (uno de ellos es el uso de un algoritmo de exponenciación).

Se llevó a cabo con la función de generar aleatoriamente los números que posee la librería NTL en una computadora con las siguientes características

- Procesador: **AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx 2.00 GHz**
- RAM**: 12.0 GB (11.0 GB usable)**

Conclusiones:

Dado las comparaciones realizadas anteriormente se puede concluir que el algoritmo más rápido evitando el teorema del resto chino es el de exponenciación binaria probablemente por el poco manejo en variables que este da y su enfoque en trabajar con los bits del numero ingresado, sin embargo, no se descarta un probable error en los cálculos a causa de un fallo en la implementación de los demás algoritmos.