Analisis Algoritmos de Exponenciación Modular

Becerra Sipiran, Cledy Elizabeth
Oviedo Sivincha, Massiel
Villanueva Borda, Harold Alejandro

Resumen:

Este documento es sobre la exponenciación modular y sus versiones, el documento muestra que hay muchos algoritmos, algunos de estos mejoran agregando algunos teoremas. Pero como vemos en el análisis de los algoritmos únicamente los evaluamos de acuerdo a su base como algoritmo, sin agregar estos teoremas.

1.Introducción:

La exponenciación modular se utiliza en criptografía de clave pública.
Implica calcular b elevado a la potencia e (mod m):

$$c \leftarrow (b\text{^}e)(\text{mod } m)$$

Puede forzar este problema multiplicando b por sí mismo e - 1 veces y tomando la respuesta mod m, pero es importante tener algoritmos rápidos (eficientes) para que este proceso tenga alguna aplicación práctica.
En criptografía, los números involucrados suelen ser muy grandes. Sin un algoritmo eficiente, el proceso llevaría demasiado tiempo.

Pequeño teorema de Fermat ([2]pág. 459):

Sea p un primo y sea x un entero tal que x mod p != 0. Entonces

$$x^{p-1} \cong 1 \ (\text{mod} p)$$

Demostración: Es suficiente demostrar el resultado para 0 <x <p, porque

$$x^{p-1} \bmod p = (x \bmod p)^{p-1} \bmod p$$

ya que puede reducir cada subexpresión "x" en el módulo "x^{p-1}"mod p.
Sabemos que para 0 <x <p, el conjunto {1; 2; … ; p − 1ga y el conjunto {x1; x2 ; .. ; x (p − 1)} contienen exactamente los mismos elementos. Entonces, cuando multiplicamos los elementos de los conjuntos, obtenemos el mismo valor, es decir, obtenemos

$$1*2 \ldots (p-1) = (p-1)!$$

En otras palabras

$$(x*1)\ (x*2) \ldots (x*(p-1)) \cong (p-1)! \ (\text{mod } p)$$

Si factorizamos los x términos, obtenemos

$$x^{p-1}\,(p-1)! \cong (P-1)! \ (\text{mod } p)$$

Ya que p es primo, cada elemento no nulo en Zp tiene un inverso multiplicativo. Por tanto, ¡podemos cancelar el término (p − 1)! desde ambos lados, dando $x^{p-1} \cong 1$ mod p, el resultado deseado.

Teorema de Euler ([2]pág. 461):
Sea un entero positivo y sea un entero tal que gcd (x; n) = 1.

$$x^{\varphi(n)} \cong 1(\text{mod } n)$$

Prueba: La técnica de prueba es similar a la del Pequeño Teorema de Fermat. Denote los elementos de Zn+, el grupo multiplicativo para Zn, como u1; u2; … ; uφ (n) .Por la propiedad de cierre de Zn+.

$$Zn+=\{xui:i=1; \ldots ;\varphi(n)\}$$

es decir, multiplicar elementos en Zn+ ´por x mod n simplemente permuta la secuencia u1; u2; … ; uφ (n). Así, multiplicando los elementos de Zn+, obtenemos:

$$(xu1), (xu2), \dots, (xu\varphi(n)) \cong u1u2 \dots u\varphi(n) (\text{mod } n).$$

Nuevamente, recolectamos un término $x^{\varphi(n)}$ en un lado, lo que nos da la congruencia

$$x(u1u2 \dots u\varphi(n)) \cong u1u2 \dots u\varphi(n) (\text{mod} n)$$

Dividiendo por el producto de la ui's, da nuestro $x^{\varphi(n)} \cong 1 \text{ mod } n$

El teorema de Euler da una expresión de forma cerrada para los inversos multiplicativos. Es decir, si x y no son primos relativos, podemos escribir

$$x^{-1} \cong x^{\varphi(n)-1} (\text{mod} n):$$

2. Algoritmos

2.1 Exponenciación modular rápida:

Enfoque matemático ([1] pág. 463):

El algoritmo de cuadratura repetida Una observación simple pero importante para un algoritmo de exponenciación mejorado es que cuadrar un número es equivalente a multiplicar su exponente p por dos. Además, multiplicar dos números a^p y a^q es equivalente a calcular $a^{(p+q)}$. Por lo tanto, escribamos un exponente p como un número binario Pb − 1 … P0, es decir,

$$P = P(b-1)\,2^{b-1} + \dots + P(0)2^0$$

Por supuesto, cada uno de los Pi's es 1 o 0. Usando la observación anterior, podemos calcular $a^p \text{ mod } n$ mediante una variación de la regla de Horner para evaluar polinomios, donde el polinomio en cuestión es la expansión binaria anterior del exponente p. Específicamente, defina qi es el número cuya representación binaria está dada por los bits más a la izquierda de p, es decir, qi está escrito en binario como P(b – 1) … P(b – i). Claramente, tenemos p = qb. Note que q1 = P(b – 1) y podemos definir qi recursivamente como:

$$qi = 2qi - 1 + P(b - i) \text{ for } 1 < i <= b:$$

Por lo tanto, podemos evaluar $a^p \text{ mod } n$ con el cálculo recursivo, llamado método de repetición cuadrática.
La idea principal de este algoritmo es considerar cada bit del exponente p dividiendo p por dos hasta que p llegue a cero, elevando al cuadrado el producto actual Qi para cada bit. Además, si el bit actual es uno (es decir, un par impar), también multiplicamos en la base a. Para ver por qué funciona este algoritmo, define, para i = 1, … , b.

$$Qi = a^{qi} \text{ mod } n$$

De la definición recursiva de qi, derivamos la siguiente definición de Qi:

$$Qi = (Q^2(i-1) \text{ mod } n)\, a^{P(b-1)} \text{ mod } n \text{ para } 1 < i <= 1$$

$$Q1 = P^{(b-1)} \text{ mod } n$$

Es fácil verificar que $Qb = a^p \text{ mod } n$.

Algoritmo:

FastExponentiation(a, p, n):
Input: Integers a, p, and n
Output: $r=a^p \text{ mod } n$
If p=0 then
  Return 1
If p is even then
  t = FastExponentiation(a, p/2, n)
  return $t^2 \text{ mod } n$
t=FastExponentiation(a, (p−1)/2, n)
return $a(t^2 \text{ mod } n) \text{mod } n$

Seguimiento Numérico:

a	p	p%2	t			
572^1	29	1	MOD(572^1,713)	572	IF(p%2=1,MOD(572*1,713),1)	572
572^2	14	0	MOD((572^1)^2,713)	630	IF(p%2=1,MOD(630*572,713),572)	572
572^4	7	1	MOD((572^2)^2,713)	472	IF(p%2=1,MOD(472*572,713),572)	470
572^8	3	1	MOD((572^4)^2,713)	328	IF(p%2=1,MOD(328*470,713)470)	152
572^16	1	1	MOD((572^8)^2,713)	634	IF(p%2=1,MOD(634*152,713),152)	113

Implementación en C++:

```
ZZ mod(ZZ a,ZZ b){
   ZZ r=a-(b*(a/b));
   if(r<0)r=b-r;
   return r;
}
bool even(ZZ a){
   if(mod(a,ZZ(2))==0) return 1;
   return 0;
}
ZZ Fast_Exponentation(ZZ a, ZZ p, ZZ n){
   ZZ t;
   if(p == 0) return ZZ(1);

   if(even(p)){
         t = Fast_Exponentation(a, p/2, n);
         return mod((t*t),n);
   }
   t = Fast_Exponentation(a, (p-1)/2, n);
   return mod((a*(mod((t*t),n))),n);

}
```

Pruebas:

Procesador/Bits	AMD Ryzen 7 4800H 2.90 GHz	Intel(R) i5-6200U 2.30GHz	Intel(R) i3-3110M 2.40GHz
16	0.026 s	0.032 s	0.029 s
32	0.038 s	0.045 s	0.056 s
64	0.071 s	0.075 s	0.125 s
128	0.135 s	0.147 s	0.193 s
256	0.269 s	0.263 s	0.424 s
512	0.522 s	0.532 s	0.668 s
1024	1.045 s	1.002 s	1.343 s
2048	2.004 s	2.177 s	3.361 s

Eficiencia:

Tenga en cuenta que, dado que el operador de módulo se aplica después de una operación aritmética en el método Fast Exponentiation, el tamaño de los operandos de cada operación de multiplicación y módulo nunca es superior a 2[log2 n] bits [2].

2.2 Exponenciación modular binaria

Descripción:

Este algoritmo recorre los bits del exponente de izquierda a derecha (o bien de derecha a izquierda, aunque en este caso se requiere una variable adicional) .

Algoritmo:

IMPUT: a, n and m = (n−1 ...n0)
OUTPUT: The element a^n mod m.
1. r = 1
2. while n different from 0
  2.1 if n is odd then
      2.2.1 a = a^2 modulo m
  2.2 n/2
4 return r

Seguimiento Numérico:

Donde de esta calculando $48^{11} MOD\ 91$.

r	a	n	m
572	630	14	713
572	412	7	713
470	328	3	713
152	634	1	713
113	537	0	713

Implementación en C++:

```
ZZ binary_expo_modular(ZZ a, ZZ n, ZZ m){
```

```
    ZZ result;
    result = ZZ(1);
    while( n != ZZ(0)) {
        if(MOD(n,ZZ(2))!=0)
            result = MOD(result*a,m);
        a = MOD(a*a,m);
        n >>= 1;
    }
    return result;
}
```

Pruebas:

Procesador/Bits	AMD Ryzen 7 4800H 2.90 GHz	Intel(R) i5-6200U 2.30GHz	Intel(R) i3-3110M 2.40GHz
16	0.019 s	0.026 s	0.025 s
32	0.022 s	0.036 s	0.036 s
64	0.060 s	0.073 s	0.101 s
128	0.087 s	0.125 s	0.186 s
256	0.175 s	0.253 s	0.357 s
512	0.339 s	0.486 s	0.665 s
1024	0.649 s	0.990 s	1.408 s
2048	1.272 s	2.092 s	2.984 s

2.3 Right-to-left binary exponentiation

Algoritmo:

INPUT: an element g ∈ G and integer e ≥ 1.
OUTPUT: g^e.
1. A←1, S←g.
2. While e 6= 0 do the following:
   2.1 If e is odd then A←A · S.
   2.2 e←be/2c.
   2.3 If e 6= 0 then S←S · S.
3. Return(A).

Seguimiento Numérico:

Teniendo en cuenta que hallaremos g^{283}

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}	–

Implementación en C++:

```
ZZ mod(ZZ a,ZZ b){
    ZZ r=a-(b*(a/b));
    if(r<0)r=b-r;
    return r;
}
bool even(ZZ a){
    if(mod(a,ZZ(2))==0) return 1;
    return 0;
}
ZZ Right_to_left_binary_exponentiation(ZZ g,
ZZ e){
    ZZ A, S;
    A=1; S = g;
    while(e != 0){
            if(!even(e)){
            A = A*S;
            }
            e = e/2;
            if (e != 0){
            S = S*S;
            }
    }
    return A;
}
```

Eficiencia:

Sea t + 1 la longitud de bits de la representación binaria de e, y sea wt(e) el número de unos en esta representación. El algoritmo Right-to-left binary exponentiation realiza t cuadraturas y wt(e) - 1 multiplicaciones. Si e se selecciona aleatoriamente en el rango 0 ≤ e <| G | = n, entonces se pueden esperar alrededor de |_logn_| elevaciones al cuadrado y 1/2 (|_lgn_| + 1). (La asignación 1 · x no se cuenta como una multiplicación, ni la operación 1 · 1 se cuenta como un cuadrado.) Si el cuadrado es aproximadamente tan costoso como una multiplicación arbitraria, entonces la cantidad

de trabajo esperada es aproximadamente 3/2 multiplicaciones |_lgn_| [1].

2.4 Left-to-right binary exponentiation

Algoritmo:

INPUT: g ∈ G and a positive integer e = $(e_t, e_{t-1}, \ldots, e_1, e_0)_2$

OUTPUT: g^e

1. A←1.

2. For i from t down to 0 do the following:

    2.1 A←A · A.

    2.2 If $e_i = 1$, then A←A · g.

3. Return(A).

Seguimiento Numérico:

Teniendo en cuenta que hallaremos g^{283} donde t =8.

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++:

```
ZZ left_to_right_binary_exponentiation(ZZ g, ZZ e){
   ZZ A;
   A=ZZ(1);
   while(e){
        if(e&1){
        A = A*g;
        }
        g = g*2;
        e>>=1;
   }
   return A;
}
```

2.5 Naive Exponentiation

Descripción:

Este algoritmo de exponenciación "naive" es correcto, pero no es muy eficiente, ya que se necesitan (p) iteraciones para calcular la exponenciación modular de un número a la potencia p. Con grandes exponentes, este tiempo de ejecución es bastante lento.

Algoritmo:

Input: Integers a, p, and n
Output: $r=a^p \bmod n$

```
r=1
for i=1 to p do:
   r=(r*a) mod n
return r
```

Seguimiento numérico:

Donde de esta calculando $48^{11} MOD\ 91$

e=11	r=1	Resultados
0	r=(1*48)MOD 91	48
1	r=(48*48)MOD 91	29
2	r=(29*48)MOD 91	27
3	r=(27*48)MOD 91	22
4	r=(22*48)MOD 91	55
5	r=(55*48)MOD 91	1
6	r=(1*48)MOD 91	48
7	r=(48*48)MOD 91	29
8	r=(29*48)MOD 91	27
9	r=(27*48)MOD 91	22
10	r=(22*48)MOD 91	55

Implementación en C++:

```
ZZ mod(ZZ a,ZZ b){
   ZZ r=a-(b*(a/b));
   if(r<0)r=b-r;
   return r;
}
ZZ naive_exponentation(ZZ a, ZZ p, ZZ n){
   ZZ r;
   r = 1;
   for(ZZ i = ZZ(0); i < p; ++i){
         r = mod((r*a),n);
   }
   return r;
}
```

Pruebas:

Procesador /Bits	AMD Ryzen 7 4800H 2.90 GHz	Intel(R) i5-6200U 2.30GHz	Intel(R) i3-3110M 2.40GHz
16	0.057 s	0.156 s	0.924 s
32	-	-	-
64	-	-	-
128	-	-	-
256	-	-	-
512	-	-	-
1024	-	-	-
2048	-	-	-

3.Análisis de Algoritmos

Los análisis de los algoritmos se realizaron en tres diferentes procesadores (podrá ver los diferentes resultados en las pruebas realizadas al final de cada descripción del algoritmo) ,pero mismo sistema operativo:

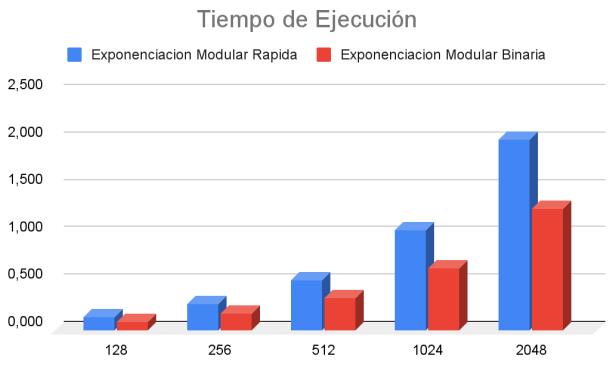
- AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz
- Intel(R) core(TM) i5-6200U CPU @ 2.30GHz 2.40GHz
- Intel(R) Core(TM) i3-3110M CPU @ 2.40GHz 2.40 GHz

Teniendo en cuenta que para el análisis tomamos el procesador más rápido de los tres; es decir , AMD Ryzen 7 4800H con Radeon Graphics 2.90 GHz.

3.1.Tiempo de Ejecución

En la siguiente tabla y gráfico mostrará el tiempo de compilación en (s) dependiendo de los bits implementados.

	Exponenciación Modular Rápida	Exponenciación Modular Binaria	Exponenciación Naive
128	0,135	0,087	∞
256	0,269	0,175	∞
512	0,522	0,339	∞
1024	1,045	0,649	∞
2048	2,004	1,272	∞



Cómo se logra ver en la gráfica la Exponenciación Modular Rápida es el algoritmo que destaca por su larga duración al compilar dependiendo de los bits llegando hasta los 2,004 s . También resalta la Exponenciación Modular Binaria por su menor tiempo de compilación.
Tener en cuenta que la Exponenciación Naive es muy primitiva y tarda demasiado para números grandes desde los 32 bits .

4.Conclusiones

Para finalizar este análisis, se puede observar el comportamiento de diferentes algoritmos para el mismo resultado u objetivo, donde comparamos su eficacia. En primer lugar, se tiene 3 algoritmos referidos a la

Exponenciación Modular ; después de evaluar su tiempo de ejecución, afirmamos en la alta eficacia de la Exponenciación modular binaria.

5. Referencias bibliográficas:

[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