# Índice

# Factorización

Wilber Cutire Sivincha

09-11-15

Los algoritmos de Factorización a lo largo de la historia ha demandado una constante investigación lo que llevo a la necesidad de Algoritmos de verificación de primalidad, y algoritmos de multiplicación rapida. Estos algoritmos han ido variando y encontramos algoritmos determinísticos, de propósito general y probabilísticos. Donde de este ultimo grupo de algoritmos nos vamos a enfocar mas para poder entender la Criba Cuadrática.

## 1. Metodo de Eratóstenes

El algoritmo de la criba de Eratóstenes es marcar como compuesto los números que son múltiplos de algún número que no ha sido tachado, entonces podemos tachar los números múltiplos por algún primo $p$ desde el cuadrado del primo ($p^2$) ya que los menores a este cuadrado han sido tachados por primos mas pequeños.

La idea de Factorización usando la criba de Eratóstenes es guardar aquel primo por el cual algun número compuesto fue tachado.

```
#define MAXI 200000002

int Criba[MAXI];

using namespace NTL;
using namespace std;

void erastotenes( vector<int>* Primes){
  Criba[4]=Criba[6]=2;
  Primes->push_back(2);
  Primes->push_back(3);
  Primes->push_back(5);
  int R[]={4,2,4,2,4,6,2,6};
  for (int i=7,cur=0;i<MAXI;i+=R[cur++&7]){
    if(!Criba[i]){
      Primes->push_back(i);
      for (int j=i*i;j<=MAXI;j+=2*i)
        Criba[j]=i;
      }
```

```
  }
  return;
}
```

Como todos los números compuestos están marcados por algún factor primo $p$, podemos representar un numero $N$ por sus factores primos buscando los factores de $N/p$ junto a $p$.

### 1.1. Algoritmo

```
void erastostenes_factorization(int N,map< int,int >* primeFactors){
  while (Criba[N]!=0){
    N/=Criba[N];
    (*primeFactors)[Criba[N]]++;
  }
  (*primeFactors)[N]++;
  return;
}
```

Usando una rueda nos acercamos a una complejidad lineal para la creación de la criba. La complejidad para sacar los factores primos ya que es amortizado por la Criba es cercana a la logaritmica haciendo uso de un vector, y $\log^2$ usando map.

### 1.2. Análisis

| bits | $p * q$ | milisegundos |
|---|---|---|
| 8 | 77 | 2.4429e-05 |
| 16 | 323 | 2.5653e-05 |
| 32 | 96721 | 8.017e-06 |
| 64 | 13867981 | 9.706e-06 |
| 128 | 1487299539970274071 | – |
| 256 | 398957659543669698902988282568401641140541333697796547285321 3 | – |

## 2. Metodo de Fermat

La idea de este algoritmo es intentar hallar enteros positivos x e y tales que:

$$n = x^2 - y^2 = (x - y)(x + y)$$

Para un “n” impar dado como entrada . Al suponer que encontramos x e y, entonces los factores de“n” serán $(x - y) * (x + y)$.

El primer paso es tomar la parte entera y aplicar la raíz cuadrada a esta, es suficiente con que nos quedemos con el valor de la parte entera de $\sqrt{n}$, entonces obtenemos un número "x". Se puede ver que el caso más fácil de factorización de fermat es que "n" sea un cuadrado perfecto, pues basta con hallar su raíz para encontrar un factor, entonces $x^2 = n$ $y = 0$ , pero si $y > 1$ se puede ver que $x = \sqrt{n + y^2}$ o que $y = sqrtx^2 - n$.

En el último caso es cuando al hallar la parte entera de $\sqrt{n = x}$ , incrementamos en 1 a este resultado, y se calcula $y = \sqrt{x^2 - n}$ . Esto es el segundo paso.

En una tercera instancia se repite los pasos del paso 2, hasta que encontremos un valor entero para $y = \sqrt{x^2 - n}$ o hasta que x sea igual a $(n + 1)/2$. En el primer caso n tiene como factores a $(x - y)$ e $(x + y)$, en el segundo caso n es primo. Veamos un ejemplo para un n=1342127

En $x$ almacenaremos la parte entera de $\sqrt{n}$, esto es $x = 1158$, luego se incrementa x en una unidad $x = 1159$ , y hallamos $y = \sqrt{x^2 - n}$ , en la tabla se pueden observar los resultados:

| x | $x^2 - n$ |
|---|---|
| 1159 | 33,97 |
| 1160 | 58,93 |
| 1161 | 76,11 |
| 1162 | 90,09 |
| 1163 | 102,18 |
| 1164 | 113 |

Como se obtuvo un entero en el sexto bucle, según el algoritmo se debe detener, $x = 1164$ e $y = 113$, entonces nuestros factores son $x + y = 1277$ e $x - y = 1051$.

## 2.1. Algoritmo

```
1  fermat_factorization( int n ){
2      int x;
3      x = sqrt(n);
4      if(n == pow(x,2.0)){
5          cout<< x << " es factor de " << n <<endl;
6      } else {
7          x=x+1;
8          float y = float(sqrt(pow(float(x),2.0)-n));
9          while(fmod(y,float(1)) != 0 && x != (n+1/2)){
10             x=x+1;
11             y=float(sqrt(pow(float(x),2.0)-n));
12         }
13         if(x == ((n+1)/2))
14             cout<< n << " es primo" << endl;
15         else
```

```
16                cout<< x+y <<","<< x-y << " son factores de n" << endl;
17        }
18 }
```

## 3. Metodo de Pollard Rho (Pollar $\rho$)

También conocido como el método de factorización Pollard Monte Carlo. Es un algoritmo de factorización de propósito específico, ya que es muy efectivo a la hora de factorizar números compuestos que tengan factores pequeños.

El algoritmo emplea una función módulo n para generar una secuencia pseudoaleatoria, la idea detrás del algoritmo de pollard rho es encontrar ciclos. Por ejemplo si se usa la ecuación para detectar los ciclos:

$$F(x) = x^2 \mod n$$

con $n = 323$ y $x = 2$, al cabo de un número finito de operaciones los resultados empiezan a repetirse:

$F(2) = 2^2 \mod 323 = 4$
$F(4) = 4^2 \mod 323 = 16$
$F(16) = 16^2 \mod 323 = 256$
$F(256) = 256^2 \mod 323 = 290$
$F(290) = 290^2 \mod 323 = 120$
$F(120) = 120^2 \mod 323 = 188$
$F(188) = 188^2 \mod 323 = 137$
$F(137) = 137^2 \mod 323 = 35$
$F(35) = 35^2 \mod 323 = \mathbf{256}$
$F(256) = 16^2 \mod 323 = \mathbf{290}$
$F(290) = 290^2 \mod 323 = \mathbf{120}$

A partir de $x = 35$ los resultados empiezan a repetirse, y dado que 35 es el primer valor en generar un resultado repetido. Lo que el algoritmo hace en su forma original es asignar a dos variables un valor, normalmente una el doble de la otra, por ejemplo $x = 2$, $y = 1$, luego cada valor es iterado en nuestra función módulo n, a una de las variables se le aplica dos veces la función módulo n, es decir su secuencia avanzara el doble de rápido, (esto es conocido como el método de floyd para la detección de ciclos) , luego cuando exista una congruencia se realiza una resta, tomando como ejemplo lo anterior (162 mód 32335), en este 16, obteniendo $35 - 16 = 19$, luego se observará si este resultado es factor de $n, d \leftarrow \gcd(|x - y|, n)$. Si el máximo común divisor es igual a "n", el algoritmo termina en fracaso (porque d=n, entonces n es primo), es decir Pollard falla si el número es primo,

pero también puede fallar en el caso de un compuesto, si es así se sugiere intentar con otra función módulo n.

Típicamente necesitamos unas $O(\sqrt{p})$ operaciones. Básicamente lo que se muestra es que, como en el problema del cumpleaños, dos números x e y , tomados de manera aleatoria, son congruentes módulo p con probabilidad mayor que 1/2, después de que hayan sido seleccionados unos 1,177$\sqrt{p}$ números. Aunque la sucesión $x_i + 1 = f(x_i)(\mod N)$ cae en un ciclo en unas $O(\sqrt{N})$ operaciones, es muy probable que detectemos $x_i \equiv x_j(\mod p)$ en unos $O(\sqrt{p})$ pasos. Si $p \approx \sqrt{N}$ entonces encontraríamos un factor de $N$ en unos $O(N^{1/4})$ pasos.

Bajo determinadas condiciones Pollard $\rho$ puede llegar a ser muy lento e ineficiente.

### 3.1. Algoritmo

```
inline ZZ rhoFactor(ZZ N) {
    ZZ x,y,d;
    x=y=2;
    d=1;
    while (d==1){
        x=f(x) %N;
        y=f(f(x2)) %N;
        d=gcd(x-y,n);
     }
    return d;
}
```

### 3.2. Análisis

| bits | $p * q$ | milisegundos |
|---|---|---|
| 8 | 77 | 4.8145e-03 |
| 16 | 323 | 8.5348e-05 |
| 32 | 96721 | 0.000190587 |
| 64 | 13867981 | 0.00047438 |
| 128 | 1487299539970274071 | 0.1016223 |
| 256 | 398957659543669698902988282568401641140541333697796547285321 3 | – |

## 4. Variante de Brent Pollard

El algoritmo usa la factorización de Monte de Carlo,también usada por pollard $\rho$, la cual es 24 % mas rápido que pollard, y usa un algoritmo de búsqueda de ciclos que es un 36 % en promedio mas rápido que Floyd que es el que usa Pollard y Pollard $\rho$.

Recordemos que en método de Pollard $\rho$ ya descrito, tenemos que buscar una secuencia que se ha convertido periódica. Por pollard usábamos la idea de Floyd de comparar $x_i$ a $x_{2i}$ para todos los i. Brent cosidera el uso de cualquier base $b$ para guardar los valores en lugar de fijar $b = 2$. Sin embargo $b = 2$ es considerado casi perfecto.

**Teorema:** Siendo $z$ es un entero positivo, $z$ es un potencia de la base 2 si y solo si la operacion $z$ AND $(z-1) = 0$. Donde $a$ AND $b$ es la operación binaria AND entre cada bit de $a$ y $b$.

```
inline bool isPowerOf2(ZZ i){
  ZZ aux=(i-1)&i;
  return aux==0? 1:0;
}
```

La mejora que se propone Richard P. Brent es sustituir el algoritmo de búsqueda de ciclos de Floyd. Con el siguiente algoritmo:

### Búsqueda de ciclos Brent Pollard(algoritmo)

1. Mantenga solo una copia en ejecución de $x_i$.
2. Si $i$ es una potencia de la base $b$, hacemos que $y = x_i$
3. En cada iteración comparamos el valor actual de $x_i$ con el valor guardado $y$.

Ya en la Factorización en lugar de comparar $x_i$ con $y$ calculamos el $\gcd(|x_i - y|, n)$.

### Algoritmo

```
ZZ brentFactor(ZZ N){
  ZZ x,y,d;
  x=y=random(N-2)+1;
  while(d==1 && d!=N){
    x = f(x) %N
    d = gcd(x - y), N)
    if (isPowerOf2(i)){
      y= x;
    }
  }
  return d;
}
```

**Teorema** Las propiedades que cumple un grupo multiplicativo de un entero compuesto $N$, también cumplen para los grupos multiplicativos de los factores de $N'$.

En teoría de numeros, un numero liso es un entero que puede factoriales completamente por numero primos pequeños. Un entero se considera **B-potencia-lisa** si todas las potencias

primas $p_i^{n_i}$ que dividen a m satisfacen:

$$p_i^{n_i} \leqslant B$$

El Algoritmo $\rho$-1 de Pollard: Es un algoritmo de factorización de proposito general, donde factoriza todos lo números del cual el numero que precede el factor, $\rho$-1, es potencia lisa.

**Primo fuerte**: En criptograma un numero $p$ es primo fuerte si satisface que:

1. $p$ es un primo grade
2. $p-1$ tiene factores primos grandes. $p = a_1 q_1 + 1$.Donde $q_1$ es un primo grande.
3. $q_1 - 1$ tiene factores primos grandes. $q_1 = a_2 q_2 + 1$.Donde $q_2$ es un primo grande.
4. $p+1$ tiene factores primos grandes.$p = a_3 q_3 - 1$.Donde $q_3$ es un primo grande.

Se ha sugerido que en la generacion de claves del RSA el modulo n debería escogerse como el producto de dos primos fuertes. Esto haría no computable la factorización de n por el algoritmo p-1 de Pollard.

Comparando la factorización con primos fuertes con el algoritmo de Gordon obtenemos los siguientes resultados.

### 4.1. Análisis

| bits | $p * q$ | milisegundos |
|---|---|---|
| 8 | 77 | 6.4781e-05 |
| 16 | 323 | 8.2775e-05 |
| 32 | 96721 | 0.000216084 |
| 64 | 13867981 | 0.00036447 |
| 128 | 1487299539970274071 | 0.0603762 |
| 256 | 39895765954366969890298828256840164114054133369779654728532l3 | – |

## 5. Criba cuadrática

La criba cuadrática s actualmente el segundo método de factorización más rapido solo después de la Criba general del cuerpo de números. Y el mas rápido en los enteros de 100 a menos dhijitos decimales, donde la implementación es mucho menos costosa que la criba de cuerpos numericos. Tal como Pollard p-1, también es un algortimo de factorización de

proposito general. Pero a diferencia de Pollard p-1 no depende de alguna estructura especial o propiedades perfectas, mas sino del tamaño del numero a ser factorizado.

El algoritmo intenta establecer un congruencia de cuadrados modulo $n$, donde es muy probable factorizar n.
**Recolección de datos**: Se recoge la información que pueda conducir a una congruencia de cuadrados.
**Procesamiento de datos**: Se coloca los datos en una matriz y lo resuelve para obtener una congruencia de los cuadrados.
Este algoritmo tiene como ventaja que es fácilmente paralizado para la recolección de datos. Donde la desventaja es el aumento desmedido de la memoria.

## 5.1. Demostración

La idea de la criba cuadrática parte de: Teniendo dos enteros $x$ e $y$, tal que $x^2 \equiv y^2($ mód $n)$, pero $x \neq y($ mód $n)$. Entonces $n|x^2 - y^2$ pero n no divide ni a $x - y$ ni $x + y$. Entonces $gcd(x-y, n)$ debe ser un factor grande de n. Partiremos de la siguiente estrategia. Dado un conjunto que contenga los primores t primos $P = \{p_1, p_2, p_3, \cdots, p_t\}$ donde cada $p_i$ de es el i-esimo primo. Buscaremos un par $(a_i, b_i)$ tal que:

1. $a_i^2 \equiv b_i \mod n$
2. $b_i = \prod_{j=1}^{t}$ , $p_j^{e_i j}$donde $e_i j \geqslant 0$. Por lo que $b_i$ es una ($p_t$-liso)potencia lisa.

Debemos encontrar un subconjunto de $b_i$s tal que su producto es un cuadrado perfecto. Por tanto todos los $e_i j$s tienen que ser par.
De un vector binario $v_i = \{v_i 1, v_i 2, \cdots, v_i t\}$ cone le vector exponente $(e_i 1, e_i 2, \cdots, e_i t)$ tal que $v_i j = e_i j \mod 2$ Si t+1 pares$(a_i, b_i)$ produce t vectores tridimensionales $v_1, v_2, \cdots, v_{t+1}$ son dependencia lineal en $\mathbb{Z}_2$.
Por lo tanto un subconjunto T incluido en $\{1, 2, \cdots, t+1\}$ tal que $\sum i \in T v_i = 0$ sobre $\mathbb{Z}_2$ y $\prod_{i \in T} b_i$ es un cuadrado perfecto. Sabemos que $\prod_{i \in T} a_i^2$ es un cuadrado perfecto tambien.

Por tanto teniendo un entero $x = \prod_{i \in T} a_i^e$ y como raiz cuadrada de $\prod_{i \in T} b_i$ obtenemos un par $(x, y)$ tal que $x^2 \equiv y^2 \mod n$y ademas $x \neq y \mod n$ entonces gcd$(x - y, n)$ es un factor grande de $n$. Si no fuese el caso continuamos reemplazando otro par del grupo.

Podemos usar nuestra Criba para saber si un número es número liso B.

```
1 void FactorOut(ZZ &N,int &p){
2   while (N %p==0){
3     N/=p;
4   }
5 }
6
7 bool isSmooth(ZZ N,vector<int>*P){
```

```
8   for (int i=0;i<P->size();i++){
9     FactorOut(N,(*P)[i]);
10  }
11  if (N==1){
12    return true;
13  }
14  return false;
15 }
```

Donde en P se guardan los primos menores B. Existe una demostración con un Análisis Heurístico donde basta calcular a B como:

$$L(n) = e^{\sqrt{ln(N)ln(ln(N)}}$$

$$B = \lceil \sqrt{L(n)} \rceil$$

## 5.2. Algoritmo

1. Generamos un numero liso B (lo más grande posible). Donde $\pi(B)$ es el numero de primos menores a B donde con el que controlaremos el espacio de memoria para la longitud de vectores y el numero de vectores necesarios.
2. Localizar $\pi(B) + 1$ numeros $a_i$ tal que $b_i = (a_i^2 \mod n)$ es un B-liso.
3. Usar un factor de $b_i$ y generar vectores exponentes modulo 2 for cada $b_i$
4. Buscar un subconjunto del vector que sumen al vector cero.
5. Multiplicar el $a_i$ junto al resultado mod n y guardar como el par $(a, b_i)$ que produce un B-potencia-lisa($b^2$).
6. Con la igualdad $a^2 = b^2 mod n$ de la cual obtenemos dos raíces cuadradas. Formando: $(a + b)(a - b)$.
7. Calculamos el $gcd(n, abs(a - b))$ siempre que no nos de 1 o n.Producimos un factor de n.

# 6. Conclusión

La desventaja factorización usando la criba de Eratóstenes o la cuadrática es la memoria. En Eratóstenes no se puede obviar pasos ya que usa la criba para factorizar, lo que hace lenta la factorizacion para números mayores a $4 * 10^8$ pasando el segundo en tiempo de ejecución pero seguro el resultado. Y en la cuadrática para guardar los valores de los residuos cuadraticos lo cual amortiza mucho en memoria cuando se guarda el residuo en mod 2.

El algoritmo de Fermat es confiable en resultado, su desventaja es que no puede factorizar números muy grandes, salvo que los primos correspondiente al número sean cercanos, así como el algoritmo p-1 de Pollard. Pero a nivel de implementaciones es difícil que se deje abierta esta posibilidad.

El algoritmo de Pollard-rho y su variante con el algoritmo de Brent son probabilísticos, es decir quizás fallen, incluso para números pequeños. Pero su potencial está en que con ambos se pueden factorizar números de mayor tamaño en un tiempo mejor que con el algoritmo de fermat.

La criba cuadrática es entre los algoritmos de factorización la mejor opción por implementación ya que la Criba general del cuerpo de números requiere un avanzado conocimiento matemático y computacional. Para poder obtener que la Criba Cuadrática pueda factorizar números hasta 100 dígitos también se requiere una mediana experiencia computacional. Aunque se puede construir una algoritmo no muy óptimo con la información presente en este informe, o aún mas sencillo usando el metodo de Jacobi.

**Referencias**

[1] Chad Seibert,Integer Factorization using the Quadratic Sieve,University of Minnesota, 2011, *http://metodos.fam.cie.uva.es/ latex/apuntes/apuntes19.pdf*

[2] Pollard, J. M. (1975), "A Monte Carlo method for factorization",*BIT Numerical Mathematics* **15**(3):331-334, doi:10.1007/bf01933667

[3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. Stein. and Clifford (2001), "Section 31.9: Integer factorization",*Introduction to Algorithms*(Second ed.), Cambridge, MA: MIT Press, pp. 896?901, ISBN 0-262-03293-7 (this section discusses only Pollard's rho algorithm).

[4] Brent, Richard P. (1980), "An Improved Monte Carlo Factorization Algorithm", BIT 20: 176-184, doi:10.1007/BF01933190

[5] Pomerance, Carl (December 1996). "A Tale of Two Sieves" . Notices of the AMS 43 (12). pp. 1473-1485.*http://www.ams.org/notices/199612/pomerance.pdf*