Cátedra No. 4 : 24 de Marzo

Transcriptor: Cristian Bravo L.

1 Algoritmo de Euclides Extendido

Recordemos de la cátedra anterior que es posible expresar la secuencia de ecuaciones del algoritmo de Euclides en forma matricial (forma que llamamos *algoritmo extendido*). Ello nos permite encontrar, dados dos enteros u y v, las soluciones enteras de la ecuación $\text{mcd}(u,v) = au + bv$. Este detalle tiene bastante importancia práctica, como veremos a continuación.

Uno de los problemas planteados en cátedras anteriores consiste en calcular el inverso multiplicativo de un elemento q dado en el conjunto $\mathcal{Z}_m$. Como sabemos que este inverso existe si y sólo si q y m son primos relativos (es decir, si $\text{mcd}(q,m) = 1$), conocemos un método para calcular este inverso, de la siguiente forma :

$$\begin{aligned} aq + bm = 1 &\Leftrightarrow [aq+bm](\bmod m) = [aq] = [a][q] = [1] \\ &\Leftrightarrow aq \equiv 1(modm) \\ &\Leftrightarrow q^{-1} \equiv a(\bmod m) \end{aligned}$$

Nos interesa saber si este cálculo puede llevarse a cabo en forma eficiente. En una cátedra anterior se probó que el algoritmo de Euclides requiere $O(\log u \log v)$ operaciones binarias, resultado que tambien es cierto para el algoritmo de Euclides extendido. Luego, basados en el resultado anterior, es posible determinar el inverso de un elemento $q \in \mathcal{Z}_m$ en $O(\log q \log m)$ operaciones binarias.

No obstante, aún no sabemos nada acerca de la complejidad real de las operaciones básicas. Es decir, si implementásemos el algoritmo extendido de Euclides en un computador tenemos una idea aproximada acerca del número de multiplicaciones y sumas que realizaría, pero no sabemos cuántas operaciones tiene que llevar a cabo un computador para dividir dos enteros, por ejemplo.

En la siguiente sección estudiaremos la complejidad de las operaciones básicas: suma, resta, multiplicación, división y mod (es decir, resto de una división entera).

2 Complejidad de Operaciones Básicas.

2.1 Suma

Básicamente, el algoritmo para sumar dos números binarios es el mismo que usamos desde niños en la escuela, sólo que nos interesa, por motivos de implementación computacional, representar los números en notación binaria. Tomemos dos enteros x e y, y representemos por z el resultado de su suma. La operación podríamos esquematizarla como sigue:

$$\begin{array}{rl} x = & x_{l-1}\dots x_0 \\ y = & \underline{y_{l-1}\dots y_0} \\ & z_l\, z_{l-1}\dots z_0 \end{array}$$

donde $x_i,\ y_i \in \{0,1\}$, $z_i = (x_i + y_i + c_i) \bmod 2$, y $c_{i+1} = \lfloor \frac{x_i+y_i+c_i}{2} \rfloor$, $i \geq 0$.

Ahora bien: para $i = 0$, $c_1 = \lfloor \frac{x_0+y_0+c_0}{2} \rfloor$, donde $c_0 = 0$, por lo que $c_1 \in \{0,1\}$. Supongamos ahora que $c_i \in \{0,1\}$. Esto nos lleva a que el mínimo valor que puede tomar c_{i+1} es 0, pues $\lfloor \frac{0+0+0}{2} \rfloor = 0$; a su vez, el máximo valor que puede tomar c_{i+1} es 1, pues $\lfloor \frac{1+1+1}{2} \rfloor = 1$. Luego, $c_i \in \{0,1\}$, $i \geq 0$; es decir, c_i es un bit. Tradicionalmente, a éste último se le llama *de acarreo*, pues representa el *desborde o acarreo* que tenemos que hacer de un grupo de números al siguiente.

Observando que $|c_i| \leq 1$, se concluye que, para sumar x e y, debemos hacer l+1 operaciones binarias de $O(1)$ cada una. Es decir, el costo total de sumar dos enteros es $O(l)$, donde l es el largo de la representación binaria del mayor de ambos números.

2.2 Multiplicación

Nuevamente, el algoritmo usado aquí es idéntico al que usamos manualmente, sólo que aplicado a números binarios. Supongamos dos enteros x e y, cuyas representaciones binarias tienen largos l y t, respectivamente. Supongamos, sin perder generalidad, que $t < l$. Podemos esquematizar la operación de multiplicación como sigue:

$$\begin{array}{ccccccc} & & x & = & x_{l-1} & \dots & x_0 \\ & & y & = & y_{t-1} & \dots & y_0 \\ \hline & & & & z_{0,l-1} & \dots & z_{0,0} \\ & & z_{1,l-1} & & \dots & z_{1,0} & \\ & \dots & & & \dots & & \\ z_{t-1,l-1} & \dots & z_{t-1,0} & & & & \\ \hline w_{t+l-1} & w_{t+l-2} & & & \dots & & w_0 \end{array}$$

donde $z_{i,j} = y_i x_j$, $w_r = \left(\sum_{i+j=r} z_{i,j} + c_r\right) \bmod 2$ y $c_{r+1} = \left\lfloor \frac{\sum_{i+j=r} z_{i,j} + c_r}{2} \right\rfloor$.

Aquí el c_r **no es** necesariamente un bit, pero se corresponde con la misma idea que el *bit de acarreo* en las fórmulas de la suma: es el *desborde* que resulta de sumar una tupla de bits.

El algoritmo es como sigue : se toma cada dígito de x, y se va multiplicando por y_0, obteniendo una tupla de bits $z_{0,l-1}\cdots z_{0,0}$, que es igual a x si $y_0 = 1$ ó igual a 0 si $y_0 = 0$. Se repite la operación para todos los dígitos de y (y_1, y_2, etc.), y se van obteniendo sucesivas cifras binarias, que vamos desplazando en un bit cada vez (ver figura). Finalmente se suman todas estas cifras.

Del análisis anterior podemos darnos cuenta de que estamos realizando lt multiplicaciones de pares de bits, y $t+l$ sumas de a lo más t bits. Esto último puede ampliarse diciendo que estamos haciendo a lo más $(t+l)(t) = t^2 + lt$ operaciones, es decir, $O(t^2 + lt) = O(lt)$, pues $t < l$. Finalmente, el multiplicar dos enteros de l y t bits de largo, lleva $O(lt + lt) = O(lt)$ operaciones binarias el ser llevada a cabo.

Como $O(lt)$ es un número relativamente grande de operaciones binarias por hacer, surge naturalmente una pregunta: ¿ Existe un algoritmo más rápido que permita multiplicar dos enteros ? La respuesta es sí, se han descubierto varios algoritmos para multiplicar enteros mejores que el descrito más arriba. Sin embargo, el problema de determinar el orden exacto de operaciones binarias que requiere una multiplicación sigue siendo un problema abierto.

Veamos uno de estos métodos, sugerido por Katsuba y Ofman en 1962 [1].

Sean dos enteros x e y, ambos representables en l bits. Supongamos, por comodidad, que l es una potencia de dos. Sean finalmente $x_0, y_0, x_1, y_1 \in \{0, \ldots, 2^{l/2} - 1\}$ tales que:

$$x = x_1 2^{l/2} + x_0 = (x_1 x_0)_{2^{l/2}}$$

$$y = y_1 2^{l/2} + y_0 = (y_1 y_0)_{2^{l/2}}$$

En general es posible hacer esto, puesto que l es potencia de dos. Cuando representamos x_i e $y_i, i \in \{0,1\}$ en notación binaria, es muy fácil determinar estos últimos: si *cortamos* la representación binaria de x en dos *trozos* de igual cantidad de dígitos, entonces el *trozo* de la izquierda será x_1 y el de la derecha será x_0; y de igual forma se obtienen y_1 e y_0. La notación empleada en las igualdades de la derecha significa que podemos representar x e y como números de dos dígitos, en base $2^{l/2}$, donde x_i e y_i son estos dígitos.

En la práctica obtener x_1, x_0, y_1 e y_0 es muy rápido si lo implementamos a nivel de hardware, pues estamos haciendo un *shift* hacia la izquierda de $l/2$ bits. El producto de ambos números es:

$$xy = (\underbrace{x_1 y_1}_{z_1})2^l + (\underbrace{x_1 y_0 + x_0 y_1}_{z_2})2^{l/2} + \underbrace{x_0 y_0}_{z_3}$$

Llamemos en la expresión anterior $z_1 = x_1 y_1$, $z_2 = x_1 y_0 + x_0 y_1$ y $z_3 = x_0 y_0$. Podemos relacionar estas nuevas variables a través de la siguiente expresión:

$$z_2 = (x_0 + x_1)(y_0 + y_1) - z_1 - z_3$$

Luego, básicamente estamos calculando xy al precio de 3 multiplicaciones: dos para z_1 y z_3, y una tercera para z_2, además de unas cuantas sumas.

Analicemos la cantidad de operaciones binarias necesarias para llegar a determinar xy, a partir de las ecuaciones de más arriba.

Como tomamos la representación binaria de x y la *cortamos* en dos trozos del mismo número de dígitos binarios, entonces la representación binaria de x_0 e x_1 tienen la mitad del largo que la de x (y análogamente con y). Luego, las multiplicaciones involucradas en el cálculo de las tres cantidades

[1] A. Karatsuba, *Doklady Akademiia Nauk* SSSR **145** (1962), 293-294.

anteriores (z_0, z_1, z_2) se realizarán sobre enteros de $l/2$ bits de largo. Estas operaciones consumen la mayor parte del tiempo que le lleva al algoritmo hacer el cálculo de xy, pero además son necesarias dos operaciones de *shifting* sobre enteros de largo $l/2$, más un número constante de operaciones de suma de enteros de a lo más l bits; ambas operaciones se realizan en un tiempo proporcional a l, por lo que en el análisis que haremos a continuación, representamos estas operaciones como un factor constante (c) por l.

Si $T(l)$ es el tiempo que necesitamos para encontrar xy, entonces:

$$\begin{aligned}
T(l) &\leq 3T(\frac{l}{2}) + cl \\
&\leq 3(3T(\frac{l}{4}) + \frac{cl}{2}) + cl = 9T(\frac{l}{4}) + \frac{3}{2}cl + cl \\
&\leq 27T(\frac{l}{8}) + \frac{9}{4}cl + \frac{3}{2}cl + cl \\
&\leq \cdots \\
&\leq cl\left(1 + (\frac{3}{2}) + (\frac{3}{2})^2 + \ldots + (\frac{3}{2})^{\log_2 l - 1}\right) + 3^{\log_2 l}T(1) \\
&= 2cl\left(\frac{\frac{3^{\log_2 l - 1}}{2} - 1}{\frac{3}{2} - 1}\right) + 3^{\log_2 l}T(1) \\
&\leq (2c + T(1))l^{\log_2 3} = O(l^{\log_2 3})
\end{aligned}$$

Con lo anterior, podemos mejorar el tiempo de ejecución del algoritmo clásico de multiplicación de $O(n^2)$ a $O(n^{1,585\ldots})$.

Una vez que uno ha entendido el algoritmo anterior, es natural preguntarse si es posible generalizarlo, en el sentido de dividirlo en un número mayor de *trozos* (más que dos). En general es posible hacerlo, siempre que se multipliquen enteros suficientemente grandes.

Supongamos ahora que tenemos x e y, representables ambos en $(r+1)n$ bits:

$$u = (u_{(r+1)n-1} \cdots u_1 u_0)_2, \qquad v = (v_{(r+1)n-1} \cdots v_1 v_0)_2$$

donde la notación anterior significa que ambos números están en notación binaria. Ambos pueden *dividirse* en $r+1$ *trozos*, de manera que queden de la siguiente forma:

$$u = U_r 2^{nr} + \cdots + U_1 2^n + U_0, \qquad v = V_r 2^{nr} + \cdots + V_1 2^n + V_0$$

donde cada U_i y V_i son números de n bits de largo. Consideremos los siguientes polinomios:

$$U(x) = U_r x^r + \cdots + U_1 x + U_0, \qquad V(x) = V_r x^r + \cdots + V_1 x + V_0,$$

y sea

$$W(x) = U(x)V(x) = W_{2r}x^{2r} + \cdots + W_1 x + W_0.$$

Ya que $u = U(2^n)$ y $v = V(2^n)$, se cumple que $uv = W(2^n)$, así que podemos calcular uv si conocemos los coeficientes de $W(x)$. El problema que antes resolvimos definiendo tres constantes, se amplía ahora a encontrar una buena forma de calcular estos coeficientes. Esto no es difícil: basta con calcular $W_0, W_1, \ldots$ en orden:

$$U(0)V(0) = W(0) = W_0, \qquad U(1)V(1) = W(1) = W_{2r} + \cdots + W_1 + W_0, \qquad \text{etc.}$$

2.3 División

La división es bastante más complicada de realizar que la multiplicación. Sean nuevamente dos enteros u y v, de largos binarios l. La idea general es la siguiente : para calcular $q = \lfloor \frac{u}{v} \rfloor$, basta con calcular $\frac{1}{v}$ con suficiente precisión y luego multiplicar ambas cantidades. Para que la división sea satisfactoria, necesitamos l bits de precisión, es decir, necesitamos precisión $\frac{c}{2^{l+1}}$, donde $c = O(1)$, pues si obtenemos w tal que:

$$\left| w - \frac{1}{v} \right| \leq \frac{c}{2^{l+1}}$$

entonces

$$\Rightarrow \left| uw - \frac{u}{v} \right| \leq c\frac{u}{2^{l+1}} < c$$

pues u tiene l bits, u es a lo más $2^l - 1$; llamemos $\hat{q} = uw$ y desarrollemos el valor absoluto:

$$-c + q \leq -c + \frac{u}{v} < \hat{q} < c + \frac{u}{v} \leq c + q + 1$$

$$\Rightarrow q \in [\hat{q} - c - 1, \hat{q} + c] = \mathbf{I}$$

Luego, basta probar con todos los $q \in \mathbf{I}$ hasta dar con una aproximación suficientemente buena, tal que:

$$0 \leq \underbrace{u - qv}_{u \bmod v} < v$$

Ahora bien: no hemos dicho nada acerca de cómo calcular $1/v$. La idea es usar el método de Newton para resolver $f(x) = \frac{1}{v} - x = 0$, lo que converge cuadráticamente y, por lo tanto, no debiera darnos una mala cota para el tiempo total de la operación.

Iteramos entonces sobre $x_{n+1} = 2x_n - vx_n^2$ con un x_0 elegido adecuadamente (?) y obtenemos $\lfloor \frac{1}{v} \rfloor$. Veremos a continuación, sin ánimo de análisis, un algoritmo descrito en *seudocódigo*, que permite calcular esta última cantidad.

```
Inverso(v)
{
```

$x \longleftarrow \frac{1}{4}\left\lfloor \frac{32}{4v_1+2v_2+v_3} \right\rfloor$
$k \longleftarrow 0$
while $(2^k < l)$ do
  $v_k \longleftarrow (0.v_1 \dots v_{2^{k+1}+3})_2$
  $x \longleftarrow \frac{1}{2^{2^{k+1}}+1} \lfloor 2^{2^{k+1}} + 1(2x - v_k x^2) \rfloor$
  $k \longleftarrow k+1$
$return(x)$
}

Se puede probar que $x \leq 2$ y $\left|x - \frac{1}{v}\right| \leq 2^{-2^k}$ al comienzo y al final del *while*. Luego,

$$\#\text{op.bin.} = (2T(2l) + cl) + (2T(l) + cl/2) + (2T(l/2) + cl/4) + \ldots + (2T(2) + c) \leq O(l^{\log_2 3})$$

de lo que concluimos que $T(n) = O(n^{\log_2 3})$. Un análisis más riguroso permitiría concluir que para calcular $q = \lfloor \frac{u}{v} \rfloor$, se requieren $O(\log u \log v)$ operaciones binarias, lo que hace equivalentes la división y la multiplicación en términos de costo de cálculo.

2.4 Resta y Módulo

No es necesario un gran análisis para notar que la resta posee el mismo costo que la suma, con igual tamaño de entrada.

Con respecto al módulo el análisis no es tampoco demasiado complejo. Sean $u, v > 0$ enteros de l bits de largo. Calcular $u \bmod v = u - \lfloor \frac{u}{v} \rfloor v$ cuesta esencialmente lo mismo que una división y una multiplicación de números de l bits de largo, así que esta es la operación más costosa de todas las que hemos visto.