3C TIC *(Edición núm. 17)* Vol.5 – Nº 2
Junio – septiembre '16, 28- 41
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.52.28-41

Recepción: 19 de noviembre de 2015
Aceptación: 17 de junio de 2016
Publicación: 29 de junio de 2016

ALGORITMO DE BOOTH EN OPERACIONES DE INVERSO MODULAR

BOOTH ALGORITHM MODULAR REVERSE OPERATIONS

Jesús Ayuso Pérez[1]

1. Compositor musical y desarrollador software. Licenciado en Ingeniería Informática por la Universidad Carlos III de Madrid (UC3M). E-mail: ayusoperez@terra.com

3C TIC *(Edición núm. 17)* Vol.5 – Nº 2
Junio – septiembre '16, 28 - 41
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529

RESUMEN

El algoritmo dado por Andrew Donald Booth en 1950, además de permitir apoyarse en la operación inversa a aquella que compone el cómputo que estamos calculando, también nos permite pivotar sobre el elemento inverso con respecto de la misma. De esa doble interpretación, en el presente documento propondremos un algoritmo para el cálculo del inverso modular basado en el concepto ideado por Booth aplicado en este caso al algoritmo de Euclides. Veremos distintas aportaciones para realizar ese cómputo, y con el añadido de apoyarnos en operaciones que igualmente ya explotan el concepto introducido por Booth.

ABSTRACT

The algorithm given by Andrew Donald Booth in 1950, besides allowing rely on inverse operation to that composing the computing we are calculating operation, also it allows us to pivot on the inverse element with respect to the same. That double interpretation propose an algorithm for calculating the inverse modular based on the concept devised by Booth applied in this case the Euclidean algorithm. We will see different contributions to make that calculation, and with added support in operations and also exploit the concept introduced by Booth.

PALABRAS CLAVE

Booth; Algoritmo; Multiplicación; Inverso; Modular.

KEYWORDS

Booth; Algorithm; Multiplication; Inverse; Modular.

3C TIC *(Edición núm. 17)* Vol.5 – Nº 2
Junio – septiembre '16, 28 - 41
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529

INTRODUCCIÓN

Como es conocido, el concepto de Booth aplicado dentro de un contexto modular permite trabajar con inversos, simplemente teniendo constancia de que, en ese punto, el elemento con el que se está trabajando en los cálculos, no es el elemento tal cual, sino su inverso con respecto a la operación algebraica con la que estamos jugando. Esto se traduce a nivel de implementación en llevar un variable auxiliar en la que dejamos constancia de ese cambio de papel, o de esa proyección algebraica del elemento, en forma de una acción según la tabla dada originariamente por Booth. Entrando en el caso que nos ocupa, tenemos que trabajar con los inversos aditivos de los elementos en lugar de tener que calcular los congruentes positivos de los mismos; ahorrándonos así ese paso, es decir, coste en cómputo.

Partiendo de esto, se van a utilizar las operaciones primitivas referenciadas: adición, sustracción... que a su vez, como se sabe, explotan la idea de Booth, para construir un método propio para el cálculo del inverso modular. Concretamente, el presente trabajo se basará en la versión binaria del algoritmo de Euclides Extendido. Dicho método está de sobra documentado, y se hace referencia en la bibliografía al trabajo de Euclides donde éste es descrito.

3C TIC (Edición núm. 17) Vol.5 – Nº 2
Junio – septiembre '16, 28 - 41
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529

MÉTODOS

En este apartado, lo primero que se hará será dar la implementación de las operaciones primitivas sobre las que nos vamos a basar. Más concretamente, vamos a dar varias versiones partiendo de las implementaciones que se describen en las referencias bibliográficas tituladas: '*Booth algorithm operations addition and subtraction*' y '*Booth algorithm modular arithmetic operations of addition and subtraction*'.

Para empezar, se verá una primera versión del algoritmo de suma basándonos, como ya se ha dicho, en la bibliografía referenciada. Tendríamos que la suma de dos número, *a* y *b*, de longitud *n*.

```
result = a;
for(int i = 0; i < n; i++) {
      if(b[i] == 1) result = successor(result, i);
}
```

Figura 1: Algoritmo de suma 1 (addition1).

Analizando el código, vemos que no existe absolutamente ninguna diferencia con la versión del algoritmo tomada de las referencias bibliográficas, es exactamente la misma. También se aprecia que hemos etiquetado la operación (al pie del código), con el nombre de *addition1*, lo destacamos porque nos referiremos a esa implementación con ese nombre, en lo que resta de documento.

En el caso de la resta de dos números, *a* y *b*, de longitud *n* sería:

```
result = a;
for(int i = 0; i < n; i++) {
      if(b[i] == 1) result = predecessor(result, i);
}
```

Figura 2: Algoritmo de resta 2 (subtraction1).

A continuación, se dará otra versión del algoritmo de sustracción. Para este caso, a diferencia de los 2 algoritmos previamente descritos, necesitaremos un comportamiento modular. El motivo de que para las anteriores versiones no fuera necesario tener en cuenta ese detalle, lo entenderemos más adelante al verlos incrustados dentro del algoritmo de Euclides Extendido.

Concebimos la resta de dos números, *a* y *b*, módulo *m*, los 3 de longitud *n*, que etiquetaremos como *mSubtraction2*, de la siguiente manera:

```
result = a;
if(b > a) {
      for(int i = 0; i < n; i++) {
```

3C TIC (Edición núm. 17) Vol.5 – Nº 2
Junio – septiembre '16, 28 - 41
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529

```
      switch(actionBooth(b[i], m[i]) {
            case ( 0 1 ):
                  result = successor(result, i);
                  break;
            case ( 1 0 ):
                  result = predecessor(result, i);
                  break;
      }
      }
} else {
      for(int i = 0; i < n; i++) {
      if(b[i] == 1) result = predecessor(result, i);
      }
}
```

Figura 3: Algoritmo de resta modular 2 (mSubtraction2).

Se mostrarán ahora otras de las implementaciones descritas en la bibliografía. Pero en esta ocasión sí haremos uso de inversos aditivos de los elementos congruentes. Éstos nos servirán para tener un comportamiento más natural en los cómputos, al trabajar siempre con elementos dentro del conjunto que conforman nuestra estructura algebraica y ahorrándonos la constante reducción para obtener el congruente positivo de un elemento negativo.

Empezaremos con la adición, veremos una primera versión del algoritmo de suma modular basándonos en la bibliografía referenciada. Tendríamos que la suma de dos números, *a* y *b*, módulo *m*, los 3 de longitud *n,* sería:

```
result = a;
for(int i = 0; i < n; i++) {
      if(b[i] == 1) result = successor(result, i);
      if(m[i] == 1) result = predecessor(result, i);
}
if(result[n + 1] == 1) {
      for(int i = 0; i < n + 1; i++)
            result[i] = ~result[i];    // OP. DE NEGADO EN ANSI
C
```

```
      result = successor(result, 0);


      return BOOTH(0, 1); // OPERACION INVERSA
} else
      return BOOTH(1, 0);   // OPERACION
```

Figura 4: Algoritmo de suma modular 3 (mAddition3).

Como se puede observar, la anterior implementación es exactamente igual que la que se expone en el trabajo '*Booth algorithm modular arithmetic operations of addition and subtraction*' con la salvedad de que devolvemos una acción de Booth, dependiendo de si el cómputo final desborda el módulo sobre el que trabajamos o no. Dependiendo de esto, el resultado será un número positivo, o será el inverso aditivo del entero en cuestión. Cabe subrayar que éste método siempre reduce, ya que es complicado saber a priori si el resultado de la adición desbordará o no el módulo en que estamos trabajando.

Vayamos ahora a la sustracción, al caso de la resta de dos número, *a* y *b* módulo *m*, de longitud *n*. Para esta versión, *mSubtraction3*, la modularidad la vamos a gestionar con una primera comprobación de si el sustraendo es mayor que el minuendo:

```
if(b > a) {
      result = b;
      for(int i = 0; i < n; i++) {
      if(a[i] == 1) result = predecessor(result, i);
      }
      return BOOTH(0, 1); // OPERACION INVERSA
} else {
      result = a;
      for(int i = 0; i < n; i++) {
            if(b[i] == 1) result = predecessor(result, i);
      }
      return BOOTH(1, 0);   // OPERACION
}
```

Figura 5: Algoritmo de resta modular 1 (mSubtraction3).

Si nos fijamos, para esta implementación, el valor del módulo *m* no interviene en los cálculos. El motivo es porque nosotros mismos gestionamos que el resultado se encuentre siempre dentro del conjunto de elementos sobre los que trabajamos con la comprobación de que, en caso de que el sustraendo sea mayor que el minuendo, intercambiamos el papel de los valores en la operación (el sustraendo pasa a ser el minuendo y viceversa). De manera que, en tal caso, devolvemos el inverso aditivo del elemento negativo que genera nuestro cómputo. Y por supuesto, devolvemos la acción de Booth acorde con ese detalle; es decir, que ese elemento debe implicar una operación inversa: en el caso que nos ocupa, de sustracción aunque se trate de un valor positivo.

Por un lado, cabe destacar que se da por sentado que, en las todas versiones anteriores se calcula el resultado del cómputo en la variable llamada *result*, retornando el valor de la misma. Salvo en los últimos algoritmos, los cuales hacen uso de elementos inversos, y en los que entendemos que retorna un valor que hace referencia a una acción de Booth y deja el resultado del cómputo en la variable llamada *result*.

Por otro lado, de cara a una buena compresión del método de cálculo del inverso modular que queremos exponer, vamos a recordar algunas de las propiedades en las que se basa la versión binaria del algoritmo de Euclides Extendido. Resaltando primero ciertas propiedades del máximo común divisor, *mcd*, de las que se vale, que son:

1. Si *a* y *b* son pares, $mcd(a, b) = 2 * mcd(a/2, b/2)$

2. Si *a* es par y *b* impar, $mcd(a, b) = mcd(a/2, b)$

3. Si *a* y *b* son impares,

    si $a > b$, entonces $mcd(a, b) = mcd(b, a - b)$

    si b $>= a$, entonces $mcd(a, b) = mcd(a, b - a)$

Y por otra parte, también haremos uso de una propiedad del inverso modular, *minv*, que reza:

4. Si *-a* es el inverso aditivo de *a* módulo *b*, $minv(a, b) = - minv(-a, b)$

    denotando con $-minv(a, b)$ el valor $-d$ siendo $d = minv(a, b)$

Ahora repasemos la tabla dada por Booth para reducir el número de operaciones necesarias, apoyándonos en la propiedad invertible de la operación con la que se construye nuestro cálculo sobre cierta estructura algebraica:

bit menos significativo	bit extra	Interpretación	Acción
0	0	intermedio cadena de 0s	ninguna
0	1	final cadena de 1s	operación
1	0	comienzo cadena de 1s	operación inversa / inverso misma operación
1	1	intermedio cadena de 1s	ninguna

Tabla 1: Tabla de acciones de Booth.

3C TIC *(Edición núm. 17)* Vol.5 – Nº 2
Junio – septiembre '16, 28 - 41
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529

Partiendo de la tabla anterior, vamos por fin a entrar en la operación que nos ocupa: el inverso modular. Nuestra primera versión del algoritmo de Booth aplicado al cálculo del inverso multiplicativo del entero, *a* módulo *m*, los 2 de longitud *n* quedaría:

```
r0 = m
r1 = a
t0 = 0
t1 = 1
while(r1 != 0) {
 if(r0[0] = 0) {
      r0 >>= 1; // lo desplazamos a la derecha 1 bit
      if(t0[0] = 1)
      t0 = addition1(t0, m);
       t0 >>= 1; // lo desplazamos a la derecha 1 bit
 } else if(r1[0] = 0) {
      r1 >>= 1; // lo desplazamos a la derecha 1 bit
      if(t1[0] = 1)
      t1 = addition1(t1, m);
      t1 >>= 1; // lo desplazamos a la derecha 1 bit
 } else {
      if(r0 > r1) {
      r0 = subtraction1(r0, r1);
            t0 = mSubtraction2(t0, t1, m);
      } else {
      r1 = subtraction1(r1, r0);
      t1 = mSubtraction2(t1, t0, m);
      }
 }
}
return t0;
```

Figura 6: Algoritmo para inverso modular 1.

En primera instancia, hemos de reparar sobre todo en el detalle que habíamos adelantado. Se ve, primeramente, que todas las operaciones son de reducción. Buena parte de ellas, son desplazamientos a nivel de bits a la derecha (desplazamiento de 1 bit), gracias a las propiedades de la operación de máximo común divisor que se utilizan en la versión binaria del algoritmo de Euclides. Y las que no, las sumas y restas, que están etiquetadas con un *1*, en nuestras implementaciones, no generan elementos fuera de nuestra aritmética, ya que, justo después de la suma contra el módulo, se divide entre 2 el resultado (se hace una operación de desplazamiento a la derecha); y en el caso de la resta, como vemos, tenemos una comprobación, *r0 > r1*, fruto del proceso de convergencia del algoritmo de Euclides. Luego de todo ello, sabemos que la única operación que podría generar elementos fuera del módulo es *mSubtraction2*, pero recordemos que nunca será así, pues ésta reduce el módulo, valiéndose del algoritmo de Booth, para trabajar siempre con los elementos congruentes positivos.

Vayamos ahora al otro algoritmo, tal vez más interesante, que queremos documentar para el cálculo del inverso multiplicativo de un entero, *a* módulo *m*, los 2 de longitud *n*, haríamos:

```
r0 = m
r1 = a
t0 = 0
t1 = 1
opBooth1 = (0, 0);    // NINGUNA ACCION AL INICIO
opBooth2 = (0, 0);    // NINGUNA ACCION AL INICIO

while(r1 != 0) {
 if(r0[0] = 0) {
      r0 >>= 1; // lo desplazamos a la derecha 1 bit
      if(t0[0] = 1)
      t0 = addition1(t0, m);
       t0 >>= 1; // lo desplazamos a la derecha 1 bit
 } else if(r1[0] = 0) {
      r1 >>= 1; // lo desplazamos a la derecha 1 bit
      if(t1[0] = 1)
      t1 = addition1(t1, m);
      t1 >>= 1; // lo desplazamos a la derecha 1 bit
 } else {
      if(r0 > r1) {
```

```
r0 = subtraction1(r0, r1);
      if(opBooth2 == ( 1 0 )) {
            if(opBooth1 == ( 1 0 )) {
                  switch(mSubtraction3(t0, t1)) {
                        case ( 1 0 ):
                              opBooth1 = ( 0 1 );
                              break;
                        default:
                              break;
                  }
            } else
                  opBooth1 = mAddition3(t0, t1, m);
      } else if(opBooth1 == ( 1 0 )) {
                  switch(mAddition3(t0, t1, m)) {
                        case ( 1 0 ):
                              opBooth1 = ( 0 1 );
                              break;
                        case ( 0 1 ):
                              opBooth1 = ( 1 0 );
                              break;
                        default:
                              break;
                  }
      } else
      opBooth1 = mSubtraction3(t0, t1);
} else {
r1 = subtraction1(r1, r0);
      if(opBooth1 == ( 1 0 )) {
            if(opBooth2 == ( 1 0 )) {
            switch(mSubtraction3(t1, t0)) {
                        case ( 1 0 ):
```

```
                                    opBooth2 = ( 0 1 )
                                    break;
                               default:
                                    break;
                          }
                     } else
                          opBooth2 = mAddition3(t1, t0, m);
               } else if(opBooth2 == ( 1 0 )) {
                          switch(mAddition3(t1, t0, m)) {
                               case ( 1 0 ):
                                    opBooth2 = ( 0 1 );
                                    break;
                               case ( 0 1 ):
                                    opBooth2 = ( 1 0 );
                                    break;
                               default:
                                    break;
                          }
               } else
               opBooth2 = mSubtraction3(t1, t0)
          }
     }
}
if(opBooth1 = ( 1 0 ))
     t0 = subtraction1(m, t0);

return t0;
```

Figura 7: Algoritmo para inverso modular 2.

Lo primero que querríamos apostillar, es que para el algoritmo que figura sobre estas líneas, entenderemos que la operación *mAddition3* deja el resultado del cálculo en la primera variable con la que es llamada, es decir, si la llamada es *mAddition3(t1, t0, m)*, el resultado

ISSN: 2254 – 6529

del cómputo queda en *t1*, y si la llamada es *mAddition3(t0, t1, m)*, el resultado queda en *t0*. Y exactamente igual para la operación *mSubtraction3*.

Lo siguiente que salta a la vista, después de analizar el código anterior, es que la lógica de actuación entre las 2 variables que almacenan una acción de Booth, se asemeja algo enrevesada, aunque en realidad no lo es tanto. Lo que ocurre es que, para la operación que nos ocupa, la de inverso modular, al tener 2 variables que pueden guardar un elemento o su inverso, la acción a realizar: la normal o su inversa, es decir: la sustracción o la adición; decíamos, es algo más complicada de saber. Además, de que en el anterior pseudocódigo queda expresada dicha lógica por duplicado. Pero como insistimos, el criterio no es tan complejo. Resumiéndose en:

1. Si la primera variable contiene un elemento inverso y la segunda también, se ha de operar por la operación correcta.
2. Si la primera variable contiene un elemento inverso y la segunda no, se ha de operar por la operación inversa.
3. Si la primera variable contiene el elemento correcto y la segunda un inverso, se ha de operar por la operación inversa.
4. Si las dos variables contienen elementos correctos, se opera por la operación correcta.

Cabe destacar, por otra parte, esa última comprobación al final del código, en caso de que al salir del bucle hayamos terminado obteniendo en elemento congruente, en lugar del elemento resultado: obtenemos su valor real con una simple resta contra el módulo sobre el que estamos trabajando.

3C TIC *(Edición núm. 17)* Vol.5 – Nº 2
Junio – septiembre '16, 28 - 41
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529

CONCLUSIONES

El aplicar el algoritmo de Booth para trabajar con elementos inversos en lugar de congruentes positivos dentro de un contexto modular, nos ofrece la posibilidad de ahorrar cálculos para la reducción en las operaciones de sustracción de las que se hacen uso para el cómputo del inverso modular. Es cierto que tiene como contrapunto la necesidad de tener que apoyarnos complementariamente en la operación de adición, la cual, además, en nuestra implementación, siempre requerirá reducir el resultado. Pero estadísticamente, haremos uso de esa operación de adición en muchas menos ocasiones que de la operación de sustracción, en lo que al algoritmo de Euclides se refiere. Por ello, es de esperar que ofrezca un rendimiento muy superior a utilizar siempre una operación de sustracción que se verá obligada a reducir cuando exista un desbordamiento de módulo.

En conclusión, la idea propuesta por Booth, se adapta mejor a los cálculos modulares, ofreciendo una solución más elegante al poder trabajar con inversos de las operaciones con las que se construye la estructura algebraica sobre la que estamos trabajando, en lugar de vernos obligados a trabajar, por ejemplo, con congruentes positivos de éstos, enteros negativos... u otras alternativas matemáticas.

REFERENCIAS BIBLIOGRÁFICAS

Booth, A. D. (1975). A method of calculating reciprocal spacings for X-ray reflections from a monoclinic crystal. *Journal of Scientific Instruments, 22*, p. 74.

Burks, A., Goldstein, H. y Von Neumann, J. (1946). Logical Design of an Electronic Computing Instrument.

Booth, A. D. y Britten, K. H. V., (1947). General Considerations in the Design of an Electronic Computer.

Booth, A. D. (1951). A signed binary multiplication technique. *Quarterly Journal of. Mechanics. & Applied Mathematics*, *4* (2), pp. 236-240.

Ayuso, J. (2015). Booth algorithm operations addition and subtraction. *3C TIC*, *4* (2), pp. 113-119.

Ayuso, J. (2015). Booth algorithm modular arithmetic operations of addition and subtraction. *3C TIC*, *4* (3), pp. 222-229.

Euclid of Alexandria. (1557), Elements, T.L. Heath's.