Escuela Profesional de Ciencia de la Computación

Comparación de GCD

Juan Miguel Ravelo Jove
Junior Gomez Contreras
Kevin Salazar Torres
Sergio Arcos Ponce
Jorge Alfredo Tito Ccahuaya

2017

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Algoritmo de Dijkstra

1.1. Definición

Dijkstra tiene el siguiente algoritmo:

$$GCD(m,n) = \begin{cases} m & \text{Si } m = n \\ GCD(m-n,n) & \text{Si } m > n \\ GCD(m,n-m) & \text{Si } m < n \end{cases}$$

Se propone los siguientes puntos [1]:

1. ¿Por qué? $d = GCD(m,n) = GCD(m-n,n)$
2. Si $d \mid m$, $d \mid n \rightarrow\ d \mid (m-n)$

Para el punto 2:

$$d = GCD(m,n) \ \rightarrow\ d \mid m \,\&\, d \mid n$$
$$if\ m > n :$$
$$m = dq_1 \,\&\, n = dq_2$$
$$\frac{m}{n} = \frac{q_1}{q_2} \ \rightarrow\ q_1 > q_2$$

Restando $m - n$:

$$m - n = d(q_1 - q_2) \ \rightarrow\ d \mid (m-n) \tag{1.1}$$

Dado que tenemos:

$$d \mid (m-n) \ \& \ d \mid n \rightarrow \ d \mid GCD(m-n,n) \tag{1.2}$$

Para el punto 1 necesitamos tener la siguiente estructura:

- $a \mid b \,\&\, b \mid a \rightarrow \ a = b$

Partiendo de la divisibilidad:

$$\begin{aligned} d =& GCD(m,n) \rightarrow d \mid m \,\&\, d \mid n \\ s =& GCD(m-n,n) \rightarrow s \mid (m-n) \quad s \mid n \end{aligned}$$

$$\begin{aligned} m - n &= sq_3 \\ n &= sq_4 \\ m &= s(q_3 + q_4) \\ m &= sq_5 \rightarrow s \mid m \end{aligned}$$

Dado que:

$$s \mid m \ \& \ s \mid n \rightarrow \ s \mid GCD(m,n) \tag{1.3}$$

Usando el concepto de la combinación lineal:

$$\begin{aligned} d = GCD(m,n) &\rightarrow d \mid m \,\&\, d \mid n \\ d &= mx + ny \\ s = GCD(m-n,n) &\rightarrow s = (m-n)x + ny \\ s = mx + n(y-x) &\rightarrow s \mid m \,\&\, s \mid n \end{aligned}$$

$$s \mid GCD(m,n) \tag{1.4}$$

Para hacer uso de la esta propiedad:

$$a \mid b \,\&\, b \mid a \rightarrow \ a = b$$

Usaremos (2) y (4):

$$d = GCD(m, n)\ \&\ s = GCD(m - n, n)$$
$$d \mid GCD(m - n, n)$$
$$s \mid GCD(m, n)$$
$$d \mid s\ \&\ s \mid d \rightarrow\ d = s$$

Por lo tanto:

$$GCD(m - n, n) = GCD(m, n) \tag{1.5}$$

Para el caso de $m < n$ por extención se repiten los pasos anteriores y obtenemos:

$$GCD(m, n - m) = GCD(m, n) \tag{1.6}$$

1.2. Implementación

1.2.1. Implementación basica:

Es tomada directamente del algoritmo de Dijkstra sin modificación:

```
#include<NTL/ZZ.h>
#include<omp.h>
#include<cstdio>
using namespace std;
using namespace NTL;
typedef ZZ nat;
nat mcd(nat &m,nat &n){
    if(m==n)
        return m;
    else if(m>n)
        return mcd(m-n,n);
    else if(n>m)
        return mcd(m,n-m);
}
int main(){
```

```
16      printf("- Simply count elapsed time (CountTime) -\n");
17      const double startTime = omp_get_wtime();
18      nat a,b,c;
19      a=4294967295;
20      b=3294967290;
21      int k,q;
22      SetBit(a,31); //Seteamos a 32
23      SetBit(b,31);
24      k=NumBits(a); //Verificamos los bits usados
25      q=NumBits(b);
26      c=mcd(a,b);
27      cout<<"mcd: "<<c<<endl;
28      cout<<"los bits de a: "<<k<<endl
29      cout<<"los bits de b: "<<q<<endl;
30      cout<<"los bytes de a: "<<sizeof(a)<<endl;
31      cout<<"los bytes de b: "<<sizeof(b)<<endl;
32      const double endTime = omp_get_wtime();
33      printf("Duration = %lf seconds\n", (endTime-startTime));
34      printf("-----------------------------------------\n");
35      return 0;
36  }
```

1.2.2. Seguimiento del algoritmo:

En la siguiente tabla vemos las restas sucesivas entre $m\,y\,n$, donde i es el numero de iteraciones hechas.

i	m	n	GCD(m,n)
1	4294967295	3294967290	GCD(m,n)
2	1000000005	3294967290	GCD(m,n)
3	1000000005	2294967285	GCD(m,n)
4	1000000005	1294967280	GCD(m,n)
5	1000000005	294967275	GCD(m,n)

Sigue en la página siguiente.

i	m	n	GCD(m,n)
6	705032730	294967275	GCD(m,n)
7	410065455	294967275	GCD(m,n)
8	115098180	294967275	GCD(m,n)
9	115098180	179869095	GCD(m,n)
10	115098180	64770915	GCD(m,n)
11	50327265	64770915	GCD(m,n)
12	50327265	14443650	GCD(m,n)
13	35883615	14443650	GCD(m,n)
14	21439965	14443650	GCD(m,n)
15	6996315	14443650	GCD(m,n)
16	6996315	7447335	GCD(m,n)
17	6996315	451020	GCD(m,n)
18	6545295	451020	GCD(m,n)
19	6094275	451020	GCD(m,n)
20	5643255	451020	GCD(m,n)
21	5192235	451020	GCD(m,n)
22	4741215	451020	GCD(m,n)
23	4290195	451020	GCD(m,n)
24	3839175	451020	GCD(m,n)
25	3388155	451020	GCD(m,n)
26	2937135	451020	GCD(m,n)
27	2486115	451020	GCD(m,n)
28	2035095	451020	GCD(m,n)
29	1584075	451020	GCD(m,n)
30	1133055	451020	GCD(m,n)
31	682035	451020	GCD(m,n)
32	231015	451020	GCD(m,n)
33	231015	220005	GCD(m,n)
34	11010	220005	GCD(m,n)
35	11010	208995	GCD(m,n)

Sigue en la página siguiente.

i	m	n	GCD(m,n)
36	11010	197985	GCD(m,n)
37	11010	186975	GCD(m,n)
38	11010	175965	GCD(m,n)
39	11010	164955	GCD(m,n)
40	11010	153945	GCD(m,n)
41	11010	142935	GCD(m,n)
42	11010	131925	GCD(m,n)
43	11010	120915	GCD(m,n)
44	11010	109905	GCD(m,n)
45	11010	98895	GCD(m,n)
46	11010	87885	GCD(m,n)
47	11010	76875	GCD(m,n)
48	11010	65865	GCD(m,n)
49	11010	54855	GCD(m,n)
50	11010	43845	GCD(m,n)
51	11010	32835	GCD(m,n)
52	11010	21825	GCD(m,n)
53	11010	10815	GCD(m,n)
54	195	10815	GCD(m,n)
55	195	10620	GCD(m,n)
56	195	10425	GCD(m,n)
57	195	10230	GCD(m,n)
58	195	10035	GCD(m,n)
59	195	9840	GCD(m,n)
60	195	9645	GCD(m,n)
61	195	9450	GCD(m,n)
62	195	9255	GCD(m,n)
63	195	9060	GCD(m,n)
64	195	8865	GCD(m,n)
65	195	8670	GCD(m,n)

Sigue en la página siguiente.

i	m	n	GCD(m,n)
66	195	8475	GCD(m,n)
67	195	8280	GCD(m,n)
68	195	8085	GCD(m,n)
69	195	7890	GCD(m,n)
70	195	7695	GCD(m,n)
71	195	7500	GCD(m,n)
72	195	7305	GCD(m,n)
73	195	7110	GCD(m,n)
74	195	6915	GCD(m,n)
75	195	6720	GCD(m,n)
76	195	6525	GCD(m,n)
77	195	6330	GCD(m,n)
78	195	6135	GCD(m,n)
79	195	5940	GCD(m,n)
80	195	5745	GCD(m,n)
81	195	5550	GCD(m,n)
82	195	5355	GCD(m,n)
83	195	5160	GCD(m,n)
84	195	4965	GCD(m,n)
85	195	4770	GCD(m,n)
86	195	4575	GCD(m,n)
87	195	4380	GCD(m,n)
88	195	4185	GCD(m,n)
89	195	3990	GCD(m,n)
90	195	3795	GCD(m,n)
91	195	3600	GCD(m,n)
92	195	3405	GCD(m,n)
93	195	3210	GCD(m,n)
94	195	3015	GCD(m,n)
95	195	2820	GCD(m,n)

Sigue en la página siguiente.

i	m	n	GCD(m,n)
96	195	2625	GCD(m,n)
97	195	2430	GCD(m,n)
98	195	2235	GCD(m,n)
99	195	2040	GCD(m,n)
100	195	1845	GCD(m,n)
101	195	1650	GCD(m,n)
102	195	1455	GCD(m,n)
103	195	1260	GCD(m,n)
104	195	1065	GCD(m,n)
105	195	870	GCD(m,n)
106	195	675	GCD(m,n)
107	195	480	GCD(m,n)
108	195	285	GCD(m,n)
109	195	90	GCD(m,n)
110	105	90	GCD(m,n)
111	15	90	GCD(m,n)
112	15	75	GCD(m,n)
113	15	60	GCD(m,n)
114	15	45	GCD(m,n)
115	15	30	GCD(m,n)
116	15	15	GCD(m,n)

Cuadro 1.1: Seguimiento del algoritmo

1.3. Implementación modificada

Del seguimiento del algoritmo, como son restas sucesivas, podemos reducir tales restas, con *modulos* sucesivos, siendo esto un Dijkstra-Euclides.

```
1 #include<NTL/ZZ.h>
2 #include<omp.h>
```

```
#include<cstdio>
using namespace std;
using namespace NTL;
typedef ZZ nat;
nat mcd(nat &m,nat &n){
    if(m==0 || n==0)
        return m+n;
    else if(m>n)
        return mcd(m%n,n);
    else if(n>m)
        return mcd(m,n%m);
}
int main(){
    printf("- Simply count elapsed time (CountTime) -\n");
    const double startTime = omp_get_wtime();
    nat a,b,c;
    a=4294967295;
    b=3294967290;
    int k,q;
    SetBit(a,31); //Seteamos a 32
    SetBit(b,31);
    k=NumBits(a); //Verificamos los bits usados
    q=NumBits(b);
    c=mcd(a,b);
    cout<<"mcd: "<<c<<endl;
    cout<<"los bits de a: "<<k<<endl;
    cout<<"los bits de b: "<<q<<endl;
    cout<<"los bytes de a: "<<sizeof(a)<<endl;
    cout<<"los bytes de b: "<<sizeof(b)<<endl;
    const double endTime = omp_get_wtime();
    printf("Duration = %lf seconds\n", (endTime-startTime));
    printf("-----------------------------------------\n");
    return 0;
}
```

1.3.1. Seguimiento del codigo

i	m	n
1	4294967295	3294967290
2	1000000005	3294967290
3	1000000005	294967275
4	115098180	294967275
5	115098180	64770915
6	50327265	64770915
7	50327265	14443650
8	6996315	14443650
9	6996315	451020
10	231015	451020
11	231015	220005
12	11010	220005
13	11010	10815
14	195	10815
15	195	90
16	15	90
17	15	0

Cuadro 1.2: Iteraciones en Dijkstra-Euclides

En ambos casos se obtiene el mismo $GCD(m, n) = 15$, comparando de solo de estas dos implementaciones el tiempo de ejecución: Relativamente el modificado es mejor que el Dijkstra normal, por lo que los datos del Dijkstra-Euclides se usaran para comparar con los otros algoritmos.

i	Modificado	Dijkstra
1	$210\mu s$	208μ
2	$208\mu s$	215μ
3	$237\mu s$	240μ
4	$208\mu s$	154μ
5	$235\mu s$	216μ
6	$209\mu s$	219μ
7	$206\mu s$	256μ
8	$235\mu s$	251μ
9	$203\mu s$	237μ
10	$206\mu s$	267μ

Cuadro 1.3: Comparación de tiempos de ejecución

Capítulo 2

Algoritmo de Euclides clasico

2.1. Definición

El algoritmo de Euclides es un método antiguo y eficaz para calcular el máximo común divisor (MCD). Fue originalmente descrito por Euclides en su obra Elementos. El algoritmo de Euclides se basa en la aplicación sucesiva del siguiente lema:
Sean $a, b, q, r \in Z$ tales que: $a = bq + r$ con $b > 0$ y $0r < b$. Entonces $mcd(a, b) = mcd(b, r)$

2.2. Algoritmo

Recordemos que $mod(a, b)$ denota el resto de la división de a por b. En este algoritmo, en cada paso $r = mod(rn+1, rn)$ donde $rn+1 = c$ es el dividendo actual y $rn = d$ es el divisor actual. Luego se actualiza $rn + 1 = d$ y $d = r$. El proceso continúa mientras d no se anule.
Datos: $a, b \in Z \ / b \neq 0$
Salida: $mcd(a, b)$

$$
\begin{aligned}
c =& |a|, d = |b|; \\
&while\ d \neq 0\ do \\
&\quad r = mod(c, d); \\
&\quad c = d; \\
&\quad d = r; \\
&return\ mcd(a, b) = |c|;
\end{aligned}
$$

2.3. Seguimiento del codigo

i	a	b
1	4294967295	3294967290
2	3294967290	1000000005
3	1000000005	294967275
4	294967275	115098180
5	115098180	64770915
6	64770915	50327265
7	50327265	14443650
8	14443650	6996315
9	6996315	451020
10	451020	231015
11	231015	220005
12	220005	11010
13	11010	10815
14	10815	195
15	195	90
16	90	15
17	15	0

Cuadro 2.1: Seguimiento del codigo

```
1  ZZ gcd(ZZ &a, ZZ &b){
2          if (b == 0)
3            return a;
4          a%=b;
```

```
5             return gcd(b, a);
6     }
```

Capítulo 3

Algoritmo Binario de Euclides

3.1. Definición

El algoritmo binario de euclides opera con la misma idea , de cambiar mcd(a,b) por el mcd de dos numeros eventualmente pequeños , solo que esta vez solo restaremos y dividimos.
Este algoritmo opera con los siguientes teoremas:

- Si a,b son pares, $mcd(a,b) = 2mcd(a/2, b/2)$.
- Si a es par y b impar o viceversa, $mcd(a,b) = mcd(a/2, b)\, o\, mcd(a, b/2)$.
- Si a,b son impares , $mcd(a,b) = mcd(|a-b|/2, r)$, donde $r = mina, b$;

```
1  Algoritmo binario para el mcd
2  Datos: a, b e Z, a >= 0, b > 0
3  Salida: mcd ( a, b )
4      g = 1;
5      while a mod 2 = 0 And b mod 2 = 0 do
6           a = quo ( a, 2 ) , b = quo ( b, 2 ) ;
7           g = 2<<g //removiendo potencias de 2
8      while a  = 0 do // Ahora, a o b es impar
9        if a mod 2 = 0 then
10           a = quo ( a, 2 )
```

```
11        else if b mod 2 = 0 then
12            b = quo ( b, 2 )
13        else ; // ambos impares
14            t = quo (| a     b | , 2 ) ;
15            if a >= b then ; // reemplazamos m x { a, b } con
                  quo (| a - b | , 2 )
16               a = t
17            else
18               b = t
19      return g*b;
```

3.2. Implementación

```
1  #include<NTL/ZZ.h>
2  #include<omp.h>
3  #include<cmath>
4  using namespace std;
5  using namespace NTL;
6
7  using nat = ZZ;
8
9  nat dividir2(nat);
10 nat multiplicar2(nat);
11 nat GCD(nat &, nat &);
12 int main(){
13    const double startTime = omp_get_wtime();
14    nat a, b,c;
15    a = 768454923;
16    b = 542167814;
17    SetBit(a,63);
18    SetBit(b,63);
19    c = GCD(a,b);
20    cout << c << endl;
```

```
21     const double endTime = omp_get_wtime();
22     printf("Duration = %lf seconds\n", (endTime - startTime));
23     return 0;
24  }
25  nat dividir2(nat  a){
26     return a >> 1 ;
27  }
28  nat multiplicar2(nat a){
29     return a << 1;
30  }
31  nat GCD(nat &a, nat &b){
32     nat result, t,g;
33     result = 0,g = 1,t = 0;
34     while(( a % 2 == 0) && (b % 2 == 0)){
35        a = dividir2(a);
36        b = dividir2(b);
37        g = multiplicar2(g);
38     }
39     while( a != 0 ){
40        if (a % 2 == 0)
41           a = dividir2(a);
42        else{
43           if(b % 2 == 0)
44              b = dividir2(b);
45           else{
46              t = dividir2( abs(a-b) );
47              if (a >= b)
48                 a = t;
49              else
50                 b = t;
51           }
52        }
53     }
54     result = g * b;
55     return result;
```

```
56 }
```

3.3. Seguimiento del algoritmo

i	a	b	t
1	4294967295	3294967290	0
2	4294967295	1647483645	0
3	1323741825	1647483645	1323741825
4	1323741825	161870910	161870910
5	1323741825	80935455	161870910
6	621403185	80935455	621403185
7	270233865	80935455	270233865
8	94649205	80935455	94649205
9	6856875	80935455	6856875
10	6856875	37039290	37039290
11	6856875	18519645	37039290
12	6856875	5831385	5831385
13	512745	5831385	512745
14	512745	2659320	2659320
15	512745	1329660	2659320
16	512745	664830	2659320
17	512745	332415	2659320
18	90165	332415	90165
19	90165	121125	121125
20	90165	15480	15480
21	90165	7740	15480
22	90165	3870	15480
23	90165	1935	15480
24	44115	1935	44115
25	21090	1935	21090

Sigue en la página siguiente.

i	a	b	t
26	10545	1935	21090
27	4305	1935	4305
28	1185	1935	1185
29	1185	375	375
30	405	375	405
31	15	375	15
32	15	180	180
33	15	90	180
34	15	45	180
35	15	15	15

Cuadro 3.1: Seguimiento del algoritmo

Capítulo 4

Algoritmo Euclides Extendido

4.1. Definición

El algoritmo de Euclides extendido permite, además de encontrar un máximo común divisor de dos números enteros $a\,y\,b$, expresarlo como la mínima combinación lineal de esos números, es decir, encontrar números enteros $s y t$ tales que $mcd(a,b) = as + bt$. Esto se generaliza también hacia cualquier dominio euclideano.

```
1  //Entrada: Valores a y b pertenecientes a un dominio euclideo
2  //Salida: Un MCD de a y b, y los valores s y t tales q
3  mcd(a,b)=as+bt;
4  r0=a,r1=b,s0=1,t0=0,t1=1,i=0;
5  Mientras ri !=0
6   divida ri-1 entre ri para obtener qi y ri+1
7   si+1 =si-1 - qi*si
8   ti+1 =ti-1 - qi*ti
9   i=i+1
10 Resultado ri-1 es el MCD de a y b
11 se expresa ri-1=a*si-1 + b*ti-1
```

4.1.1. Fundamentos

- Usar el algoritmo tradicional de Euclides. En cada paso, en lugar de a dividido entre b es q y de resto r se escribe la ecuación $a = bq + r$

- Se despeja el resto de cada ecuación.

- Se sustituye el resto de la última ecuación en la penúltima, y la penúltima en la antepenúltima y así sucesivamente hasta llegar a la primera ecuación, y en todo paso se expresa cada resto como combinación lineal.

4.2. Implementación

```
1  nat mcd(nat r1, nat r2){
2    nat s1, s2, t1, t2;
3    s1=1; s2=0; t1=0; t2=1;
4    nat s,r,t,q;
5    s=0,r=0,t=0,q=0;
6    while(r2>0){
7      q=r1/r2;
8      r=r1-q*r2;
9      r1=r2;
10     r2=r;
11     s=s1-q*s2;
12     s1=s2;
13     s2=s;
14     t=t1-q*t2;
15     t1=t2;
16     t2=t;
17   }
18   return r1;}
```

4.3. Seguimiento del algoritmo

r1	r2	r	q	s	s1	s2	t	t1	t2
4294967295	3294967290	0	0	0	1	0	0	0	1
3294967290	1000000005	1000000005	1	1	0	1	-1	1	-1
1000000005	294967275	294967275	3	-3	1	-3	4	-1	4
294967275	115098180	115098180	3	10	-3	10	-13	4	-13
115098180	64770915	64770915	2	-23	10	-23	30	-13	30
64770915	50327265	50327265	1	33	-23	33	-43	30	-43
50327265	14443650	14443650	1	-56	33	-56	73	-43	73
14443650	6996315	6996315	3	201	-56	201	-262	73	-262
6996315	451020	451020	2	-458	201	-458	597	-262	597
451020	231015	231015	15	7071	-458	7071	-9217	597	-9217
231015	220005	220005	1	-7529	7071	-7529	9814	-9217	9814
220005	11010	11010	1	14600	-7529	14600	-19031	9814	-19031
11010	10815	10815	19	-284929	14600	-284929	371403	-19031	371403
10815	195	195	1	299529	-284929	299529	-390434	371403	-390434
195	90	90	55	-16759024	299529	-16759024	21845273	-390434	21845273
90	15	15	2	33817577	-16759024	33817577	-44080980	21845273	-44080980
15	0	0	6	-219664486	33817577	-219664486	286331153	-44080980	286331153

Capítulo 5

Algoritmo Lehmer GCD

5.1. Algoritmo

Primero es importante definir una base sobre la cual se trabajará, tomaremos 1000 como base. Esto significa que los dígitos de los números a tratar serán agrupados en conjuntos de 3; por ejemplo, si tenemos 657343812 y 431056703 se formarán grupos de (657)(343)(812) y (431)(056)(703), y se iterará sobre el grupo más significativo, sin importar que este grupo tenga 3, 2, ó 1 elemento, estos valores se almacenan en x_ e y_, pero nos aseguraremos que estas iteraciones se den siempre y cuando la cantidad de grupos restantes sea la misma (linea 1.3), si no x_ y y_ intercalarán valores por largo tiempo. Así cada vez se trabaja sobre un x e y más pequeño, a su vez estos estarán disminuyendo a lo largo de la ejecución, cuando uno de ellos sea menor o igual a la base se podrá proceder con otro método de cálculo del gcd.[2]
INPUT: Dos enteros positivos(x $\geq$ y) que sean >= a la base.
OUTPUT: gcd(x, y)

1 While y $\geq$ b do the following:

1.1 Set x_, y_ to be the high-order digit of x, y, respectively (y_ could be 0)

1.2 $A \leftarrow 1$, $B \leftarrow 0$, $C \leftarrow 0$, $D \leftarrow 1$

1.3 If ($grupos\_x == grupos\_y$) esto se agregó al algoritmo del presente libro

1.4 While $(y\_ + C)\ != 0$ and $(y\_ + D)! = 0$ do the following:

- $q \leftarrow (x\_ + A)/(y\_ + C)$, $q\_ \leftarrow (x\_ + B)/(y\_ + D)$
- if $q = q\_$ then go to step 1.5
- $t \leftarrow A - qC, A \leftarrow C, C \leftarrow t, t \leftarrow B - qD, B \leftarrow D, D \leftarrow t$
- $t \leftarrow x\_ - qy\_, x\_ \leftarrow y\_, y\_ \leftarrow t$

1.5 If $B = 0$, then $T \leftarrow xmody$, $x \leftarrow y, y \leftarrow T$

- otherwise, $T \leftarrow Ax + By, u \leftarrow Cx + Dy, x \leftarrow T, y \leftarrow u$

2 Compute $v = gcd(x, y)$ using Algorithm 2.104

3 Return(v)

5.2. Implementación

Para determinar los grupos de cifras del número se ha empleado un arreglo que contiene las potencias de la base (1000, 1000, 1 000 000, 1 000 000 000, 1 000 000 000), con el primer y último valor repetido, para que con búsqueda binaria se determine el valor adecuado con el que se hará la división para determinar x_ e y_. Por ejemplo: Al buscar 12 345 678 en el arreglo se retornará el índice de 1 000 000 y simplemente se hace 12 345 678 / 1 000 000 para obtener el 12 buscado, el índice sirve para determinar la cantidad de grupos restantes, en este caso 2. Una vez simplificado lo suficiente se emplea Dijkstra para los números reducidos.

```
1 #include <iostream>
2 #include <limits.h>
3
4 #define MAX_POTENCIAS 3 // para generar un arreglo de
     potencias de la base
5
6 using namespace std;
7 using Tipo = unsigned int;
8
9 void lehmer_gcd(Tipo x, Tipo y);
```

```
10 Tipo dijkstra_euclides( Tipo a,  Tipo b);
11
12 int main()
13 {
14     double segs;
15
16     Tipo x = 657343812;
17     Tipo y = 431056703;
18
19     cout << "Aplicando Lehmer, gcd(" << x << ',' << y << ")\n"
          ;
20     lehmer_gcd(x,y);
21
22     int dj;
23     clock_t t_ini = clock();
24     dj = dijkstra_euclides(x,y);
25     clock_t t_fin = clock();
26     cout << dj << " en " << (double)(t_fin - t_ini)*1000.0 /
          CLOCKS_PER_SEC;
27     return 0;
28 }
29
30 Tipo *genera_array_base(Tipo base);
31 Tipo digitos_base(Tipo num, Tipo arr[], Tipo &grupos);
32
33 void lehmer_gcd(Tipo x, Tipo y)
34 {
35     clock_t t_ini = clock();
36
37     Tipo x_, y_, a, b, c, d;
38     Tipo q, q_, t, tt, u;
39     Tipo grupos_x;
40     Tipo grupos_y;
41     Tipo base = 1000; // determina un arreglo de potencias de
          la base.
```

```
42      Tipo length_bits = sizeof(int)*CHAR_BIT;
43      Tipo *arr_potencias = genera_array_base(base); // se crea
           un array con las potencias de la base.
44                                              // si es base 2 se
                                                   trata de una forma
                                                   especial (moviendo
                                                   bits)
45      while(y >= base){
46
47          x_ = digitos_base(x, arr_potencias, grupos_x ); // x_
               tendr  los d gitos m s significativos que unidos
                ser n <= a la base.
48                                  // la funci n usa b squeda
                                       binaria en el array de
                                       potencias  de la base
49          y_ = digitos_base(y, arr_potencias, grupos_y);
50
51          a = 1; b = 0; c = 0; d = 1;
52          if(grupos_x == grupos_y){ //necesario pues en la
               siguiente vuelta hay que asegurar que x e y tengan
               la misma cantidad d cifras
53              while( ((y_+c) != 0) && ((y_+d) != 0) ){
54                  q = (x_+a) / (y_+c);
55                  q_ = (x_+b)/ (y_+d);
56                  if (q != q_){
57                      break;
58                  }
59                  t=a-q*c;
60                  a = c;
61                  c = t;
62                  t = b - q*d;
63                  b = d;
64                  d = t;
65                  t = x_ - q*y_;
66                  x_ = y_;
```

```
67                  y_ = t;
68              }
69          }
70          if (b == 0){
71              tt = x%y;
72              x = y;
73              y = tt;
74          }
75          else{
76              tt = a*x + b*y;
77              u = c*x + d*y;
78              x = tt;
79              y = u;
80          }
81      }
82
83      clock_t t_fin = clock();
84
85      cout << "Lehmer redujo a gcd(" << x << ',' << y << ") en "
           ;
86      double segs = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
87      cout << segs * 1000.0 << " milisegundos. \n" << '\n';
88
89      cout << "Aplicando euclides(dijkstra): gcd(" << x << ','
           << y << ") = ";
90
91      t_ini = clock();
92      Tipo v = dijkstra_euclides(x,y);
93      t_fin = clock();
94      cout << v << '\n';
95      segs += (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
96      cout << "\nTiempo total:" << segs * 1000.0 << "
           milisegundos" << '\n';
97  }
98
```

```
99  Tipo *genera_array_base(Tipo base)
100 {
101     //repetir el primer y  ltimo  elemento para
102     //  retornar lo deseado en la busq. binaria
103     //  as  se agregan dos elementos m  s
104     Tipo *arr = new Tipo[MAX_POTENCIAS+2];
105
106     arr[0] = base;
107     Tipo potencia = 1;
108     for(Tipo i = 1; i != MAX_POTENCIAS+1; i++){
109         potencia *= base;
110         arr[i] = potencia;
111     }
112     arr[MAX_POTENCIAS+1] = potencia;
113
114     return arr;
115 }
116
117 Tipo b_binaria(Tipo num, Tipo arr[], Tipo low, Tipo high); //
       devuelve el valor de la
118                                                          //
                                                               potencia
                                                                <= num
                                                                dentro
                                                                de arr
119 Tipo digitos_base(Tipo num, Tipo arr[], Tipo &grupos)
120 {
121     grupos = b_binaria(num, arr, 0, MAX_POTENCIAS+2 -1);
122     if (grupos == 0)
123         grupos = 1;
124     else
125         if (grupos == MAX_POTENCIAS+1 )
126             grupos = MAX_POTENCIAS;
127     return num / arr [grupos];
128 }
```

```
129 Tipo b_binaria(Tipo x, Tipo arr[], Tipo low, Tipo high)
130 {
131     Tipo medio;
132     if (high > low)
133         medio= (high-low)/2 + low;
134     else
135         medio = (low-high)/2 + low;
136     if(arr[medio] == x || (low > high))
137         return (low-1); //medio == low, asi retorna la
                potencia menor de la base
138     else{
139         if (arr[medio] < x)
140             return b_binaria(x,arr,medio+1, high);
141         else
142             return b_binaria(x,arr,low,medio-1);
143     }
144 }
145 Tipo dijkstra_euclides( Tipo a,  Tipo b){
146     if(a==0 || b==0)
147         return a+b;
148     else if(a>b){
149         a%=b;
150         return dijkstra_euclides(a,b);
151     }
152     else if(b>a){
153         b%=a;
154         return dijkstra_euclides(a,b);
155     }
156 }
```

5.3. Seguimiento del algoritmo

x	y	grupos_x	grupos_y	x_	y_	a	b	c	d	q	q_
657343812	431056703	2	2	657	431	1	0	0	1	1	1
						0	1	1	-1		
				431	226	1	-1	-1	2	1	1
						1	-1	-1	2		
				226	205	-1	2	2	-3	1	1
				205	21					8	11
204769594	21517515	2	2	204	21	1	0	0	1	9	9
						0	1	1	-9		
				21	15					1	3
21517515	11111959	2	2	21	11	1	0	0	1	2	1
11111959	10405556	2	2	11	10	1	0	0	1	1	1
						1	-1	-2	3		
				191	133					1	1
						-2	3	3	-4		
				133	58					2	2
						3	-4	-8	11		
				58	17					6	1
55553	23830	1	1	55	23	1	0	0	1	2	2
						0	1	1	-2		
				23	9					2	3
23830	7893	1	1	23	7	1	0	0	1	3	2
7893	151										

Capítulo 6

Comparación de algoritmos

6.1. Tiempo de ejecución

6.1.1. Dijkstra

	32b	16b	8b
1	$210\mu s$	$67\mu s$	$123\mu s$
2	$208\mu s$	$149\mu s$	$66\mu s$
3	$237\mu s$	$63\mu s$	$66\mu s$
4	$208\mu s$	$51\mu s$	$49\mu s$
5	$235\mu s$	$106\mu s$	$57\mu s$
6	$209\mu s$	$51\mu s$	$62\mu s$
7	$206\mu s$	$74\mu s$	$70\mu s$
8	$235\mu s$	$46\mu s$	$141\mu s$
9	$203\mu s$	$50\mu s$	$47\mu s$
10	$206\mu s$	$47\mu s$	$57\mu s$

Cuadro 6.1: Eficiencia en ejecución

6.1.2. Euclides Clasico

	32b	16b	8b
1	$195\mu s$	$193\mu s$	$228\mu s$
2	$1981\mu s$	$271\mu s$	$231\mu s$
3	$587\mu s$	$596\mu s$	$226\mu s$
4	$213\mu s$	$210\mu s$	$223\mu s$
5	$621\mu s$	$601\mu s$	$225\mu s$
6	$218\mu s$	$213\mu s$	$229\mu s$
7	$214\mu s$	$186\mu s$	$217\mu s$
8	$552\mu s$	$219\mu s$	$210\mu s$
9	$550\mu s$	$205\mu s$	$211\mu s$
10	$561\mu s$	$213\mu s$	$219\mu s$

Cuadro 6.2: Eficiencia de ejecución

6.1.3. Binario Euclides

	32b	16b	8b
1	$232\mu s$	$604\mu s$	$220\mu s$
2	$582\mu s$	$572\mu s$	$224\mu s$
3	$644\mu s$	$608\mu s$	$576\mu s$
4	$224\mu s$	$212\mu s$	$220\mu s$
5	$228\mu s$	$225\mu s$	$608\mu s$
6	$203\mu s$	$217\mu s$	$219\mu s$
7	$229\mu s$	$211\mu s$	$619\mu s$
8	$228\mu s$	$220\mu s$	$694\mu s$
9	$206\mu s$	$218\mu s$	$612\mu s$
10	$579\mu s$	$567\mu s$	$218\mu s$

Cuadro 6.3: Eficiencia de ejecución

6.1.4. Euclides extendido

	32b	16b	8b
1	$195\mu s$	$180\mu s$	$234\mu s$
2	$167\mu s$	$145\mu s$	$168\mu s$
3	$157\mu s$	$147\mu s$	$178\mu s$
4	$205\mu s$	$175\mu s$	$157\mu s$
5	$169\mu s$	$228\mu s$	$148\mu s$
6	$163\mu s$	$146\mu s$	$167\mu s$
7	$172\mu s$	$154\mu s$	$175\mu s$
8	$191\mu s$	$182\mu s$	$190\mu s$
9	$172\mu s$	$178\mu s$	$175\mu s$
10	$168\mu s$	$163\mu s$	$171\mu s$

Cuadro 6.4: Eficiencia de ejecución

6.1.5. Lehmer GCD

	32b	16b	8b
1	$642\mu s$	$484\mu s$	$440\mu s$
2	$1077\mu s$	$964\mu s$	$530\mu s$
3	$1135\mu s$	$461\mu s$	$537\mu s$
4	$624\mu s$	$803\mu s$	$557\mu s$
5	$640\mu s$	$467\mu s$	$453\mu s$
6	$676\mu s$	$465\mu s$	$449\mu s$
7	$696\mu s$	$491\mu s$	$948\mu s$
8	$621\mu s$	$474\mu s$	$2261\mu s$
9	$609\mu s$	$468\mu s$	$441\mu s$
10	$1168\mu s$	$456\mu s$	$447\mu s$

Cuadro 6.5: Eficiencia de ejecución

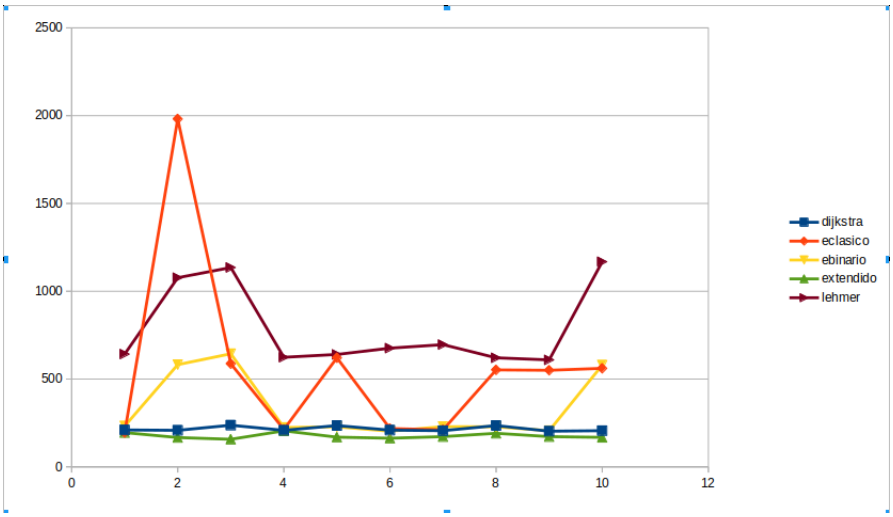

Figura 6.1: Tiempo de ejecución de 32-Bits

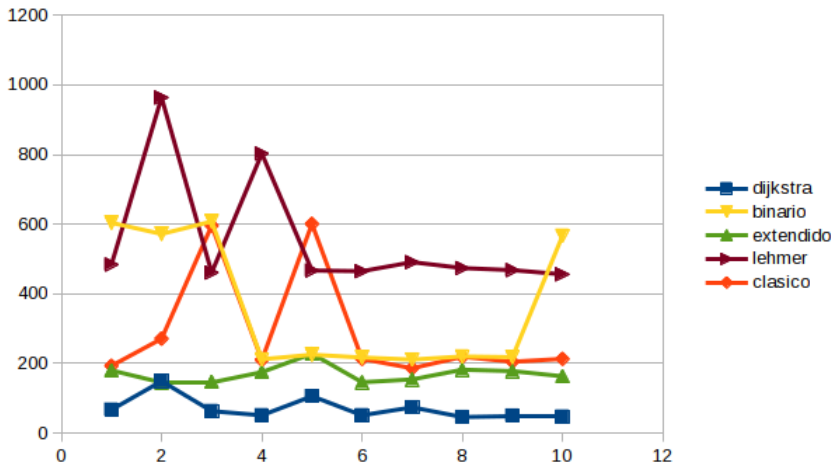

Figura 6.2: Tiempo de ejecución de 16-Bits

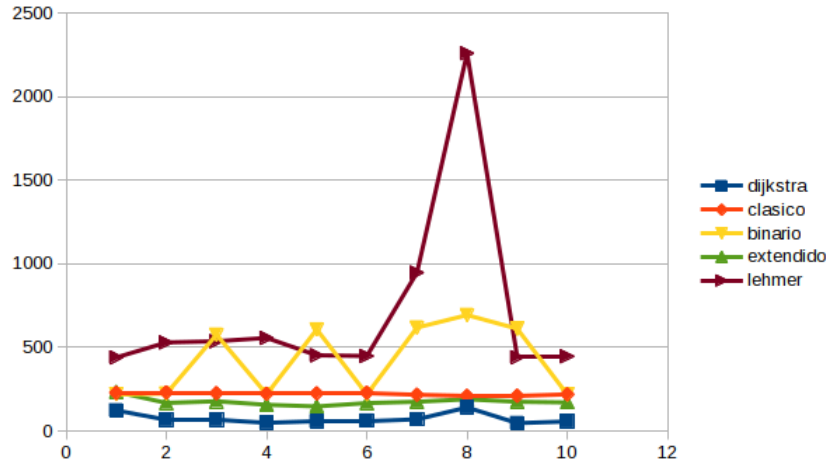

Figura 6.3: Tiempo de ejecución de 8-Bits

	dijkstra	clasico	binario	extendido	lehmer
8-bits	73.8μs	221.9μs	421μs	176.2μs	706.3μs
16-bits	70.4μs	290.7μs	365.4μs	169.8μs	553.3μs
32-bits	215.7μs	569.2μs	335.5μs	175.9μs	788.8μs

Cuadro 6.6: Promedio del tiempo de ejecución

6.2. Convergencia

	32b	16b	8b
Dijkstra	17	8	8
E. Clasico	17	8	8
E. Binario	36	21	12
E. Extendido	17	8	8
Lehmer	24	24	24

Cuadro 6.7: Numero de iteraciones por algoritmos

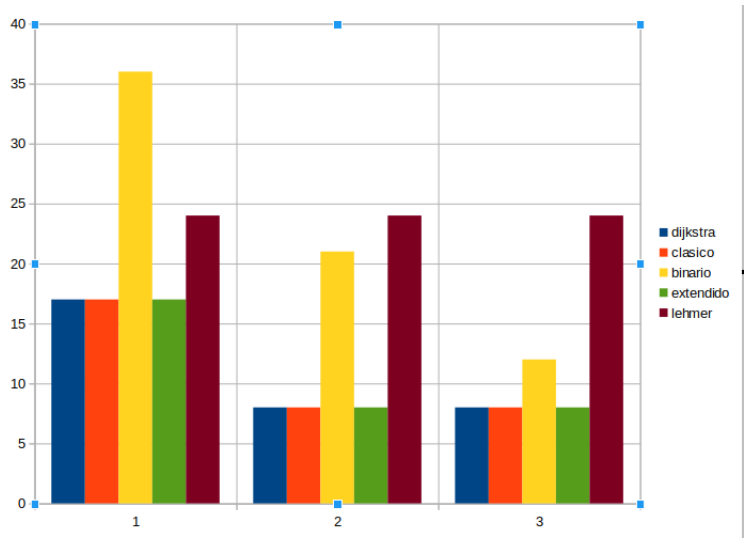

Figura 6.4: Nro de Iteraciones en 32, 16 y 8 bits

6.3. Acumulación de variables

	32b	16b	8b
Dijkstra	50	50	50
E. Clasico	44	44	44
E. Binario	5	5	5
E. Extendido	10	10	10
Lehmer	23	23	23

Cuadro 6.8: Acumulación de variables por algoritmos

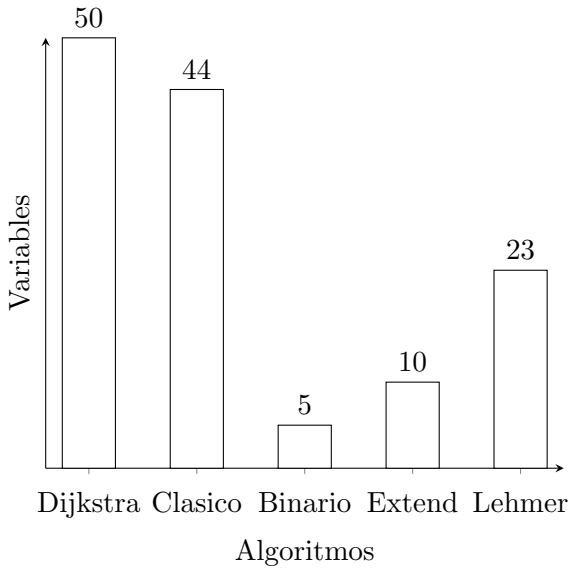

Figura 6.5: Acumulación de variable de todos los algoritmos

Capítulo 7

Conclusiones

Tiempo de ejecución

- El algoritmo de *Lehmer* tiende a un tiempo de ejecución de $500\mu s$, para cualquier cantidad de bits.
- El tiempo de ejecución del resto de algortimos es directamente proporcional al número de bits.
- El algoritmo de *Euclides Binario* presenta mayor inestabilidad(*picos en el grafico*) en el tiempo de ejecución.

Número de iteraciones

- EL algoritmo de *Dijkstra(sin modificar)* tiene el mayor número de iteraciones.
- El algoritmo de *Lehmer* sus iteraciones son independientes del número de bits.
- El resto de algoritmos son directamente proporcionales al número de bits.

Acumulación de variables

- Los algoritmos son independientes en cuanto a la acumulación de varibles.
- La mayor cantidad de variables esta reflejado en el algoritmo de Dijkstra.
- La menor cantidad de variables esta en el Euclides Binario.

Bibliografía

[1] Haroon Altarawneh. A comparison of several greatest common divisor (gcd) algorithms. *International Journal of Computer Applications (09758887)*, 26(5), 2011.

[2] Alfred J Menezes, Paul C Van Oorschot, y Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.