

The Euclidean Algorithm

Michael Monagan, monagan@inf.ethz.ch

This worksheet is intended to show two things, firstly, how to write simple programs in Maple, and secondly, a expository study of Euclid's algorithm and how to compute the greatest common divisor of two integers a and b . First, what is the greatest common divisor of two integers and why is this calculation important? Consider the following calculation that Maple does

```
> 64 / 20;
```

$$\frac{16}{5}$$

Maple has simplified the fraction $64/20$ by computing the largest integer g that divides both the numerator 64 and the denominator 20 , and cancelled it out. In this example, $g = 4$. g is the greatest common divisor or Gcd. One way to compute the Gcd of two integers is to simply to factor them and look for all the common divisors. For example

```
> 64 = ifactor( 64 );
```

$$64 = (2)^6$$

```
> 20 = ifactor( 20 );
```

$$20 = (2)^2 (5)$$

It is obvious from the factorizations that the $\text{Gcd}(64,20) = 2^2 = 4$. Unfortunately, this is a real dumb way to compute the Gcd because integer factorization is horribly expensive when you have bigger numbers. These days, the fastest computers using the best known methods can only factor numbers of about 100 digits. But we will give an algorithm here that can compute the Gcd of two integers of tens of thousands of digits quite quickly, even on your computer.

Euclid's algorithm dates back to approximately 300 BC and is perhaps the oldest algorithm in Mathematics. Euclid did not have a programming language in which to describe his algorithm. He described it in about a page of Greek text. I hope that what follows is not Greek to you! The algorithm is really simple. It assumes that the following statements are true. Let a and b be nonnegative integers.

```
> Gcd(a,0) = a;
```

$$\text{Gcd}(a, 0) = a$$

```
> Gcd(a,b) = Gcd(b,a);
```

$$\text{Gcd}(a, b) = \text{Gcd}(b, a)$$

```
> Gcd(a,b) = Gcd(a-b,b);
```

$$\text{Gcd}(a, b) = \text{Gcd}(a - b, b)$$

The first two I hope you believe. The last one needs a little proof. Let $g = \text{Gcd}(a,b)$ and $h = \text{Gcd}(a-b,b)$. The proof goes in two steps. First we show that g divides h . Then we show that h divides g . If that true, then g must be equal to h . In the proof we use the symbol --- as shorthand for "divides".

(i) (Show $g \text{---} h$). $g = \text{Gcd}(a,b)$ by definition therefore g divides a and b . Consider $a-b$. If $g \text{---} a$ and $g \text{---} b$ then we can write

$$a - b = g^*(a/g) - g^*(b/g) = g^*(a'-b')$$

i.e. $g \mid a-b$. But if $g \mid b$ and $g \mid (a-b)$ then g is a COMMON divisor of $a-b$ and b . Therefore g must divide the $\text{Gcd}(a-b, b)$. Therefore $g \mid h$ because h is the GREATEST common divisor of $a-b$ and b .

(ii) (Show $h \mid g$). $h = \text{Gcd}(a-b, b)$ hence $h \mid a-b$ and $h \mid b$. Therefore h must also divide a . Therefore h is a COMMON divisor of a and b hence $h \mid g$.

Now, Euclid's algorithm is based on this fact that $\text{Gcd}(a, b) = \text{Gcd}(a-b, b)$. If $a \geq b$, then we want to use this theorem to reduce the size of the integer a . If $a < b$, since $\text{Gcd}(a, b) = \text{Gcd}(b, a)$, we just interchange a and b . Keep doing this until one of a or b become 0. Then we stop because $\text{Gcd}(a, 0) = a$. This algorithm is most easily expressed recursively as follows:

```
> GCD := proc(a,b)
>   if b = 0 then
>     a;
>   elif a < b then
>     GCD(b,a);
>   else GCD(a-b, b);
>   end if;
> end;
```

Let's see the algorithm work on $a = 64$ and $b = 20$.

```
> GCD(64, 20);
```

4

Now I'm just going to tell Maple to print out what is happening when the Gcd program is executed so you can see the computation occurring.

```
> printlevel := 1000;
                               printlevel := 1000
> GCD(64,20);
{--> enter GCD, args = 64, 20
{--> enter GCD, args = 44, 20
{--> enter GCD, args = 24, 20
{--> enter GCD, args = 4, 20
{--> enter GCD, args = 20, 4
{--> enter GCD, args = 16, 4
{--> enter GCD, args = 12, 4
{--> enter GCD, args = 8, 4
{--> enter GCD, args = 4, 4
{--> enter GCD, args = 0, 4
{--> enter GCD, args = 4, 0
<-- exit GCD (now in GCD) = 4}
```

```

<-- exit GCD (now in GCD) = 4}
<-- exit GCD (now in GCD) = 4}
<-- exit GCD (now in GCD) = 4}
<-- exit GCD (now in GCD) = 4}
<-- exit GCD (now in GCD) = 4}
<-- exit GCD (now in GCD) = 4}
<-- exit GCD (now in GCD) = 4}
<-- exit GCD (now in GCD) = 4}
<-- exit GCD (now in GCD) = 4}
<-- exit GCD (now at top level) = 4}

```

4

There are some things we'd like to do to correct and simplify our Maple code here. We should check that the inputs a and b are nonnegative integers. This can be done this way

```

> GCD := proc(a::nonnegint, b::nonnegint)
>   if b = 0 then
>     a;
>   elif a < b then
>     GCD(b, a);
>   else GCD(a - b, b)
>   end if;
>   end;
>   printlevel := 1;

```

printlevel := 1

```

> GCD( 64, 20 );

```

4

The efficiency of the algorithm now needs to be improved. Think about how long the algorithm will take if we give it the numbers $a = 10^{10}$ and $b = 10$. What will happen? It will repeatedly subtract 10 from 10^{10} until it gets to 0. This will take 10^{10} steps which will take a very long time. Don't try it. What the algorithm really does is keep on subtracting b from a until $a < b$. In other words, it computes the remainder of a divided by b . We can do this repeated subtraction more efficiently by computing one long integer division. In Maple the functions `irem` and `iquo` compute the remainder r and the quotient q of a divided by b such that $a = b*q + r$ and $r < b$. So instead of using the theorem $\text{Gcd}(a,b) = \text{Gcd}(a-b,b)$ we can use the theorem $\text{Gcd}(a,b) = \text{Gcd}(\text{irem}(a,b),b)$ and get a much more efficient version of Euclid's algorithm. Namely

```

> GCD := proc(a::nonnegint, b::nonnegint)
> if b = 0 then
> a;
> elif a < b then
> GCD(b, a);
> else GCD( irem(a, b), b );
> end if;
> end:
> GCD( 10^10, 10 );

```

10

```

> GCD( 1234567890, 9876543210 );

```

90

It turns out that Maple uses Euclid's algorithm almost exactly as we have described it. The program is not recursive though. It is programmed in a while loop. So here is another version of Euclid's algorithm which uses a while loop.

```

> GCD := proc(a,b) local c,d,r;
> c := a; d := b;
> while d > 0 do
> r := irem(c,d);
> c := d;
> d := r;
> end do;
> c;
> end:
> GCD( 64, 20 );

```

4

A computer scientist would be interested in how efficient this algorithm really is. How many steps does it take and what is the total cost? In computer science, we have developed a methodology for talking about the efficiency of an algorithm that is independent of the particular computer and independent of the programming language. The idea is to measure the number of "atomic" operations, i.e. operations which take a constant (bounded) amount of time. In our case, the inputs a , and b are integers. Let us suppose that a and b have $\leq n$ decimal digits. What we would like is a count on the number of operations as a function of n . Euclid's algorithm turns out to be an $O(n^2)$ algorithm. This means that the number of operations is $< c \cdot n^2$ for some constant c . What this means in practice is that as the size of the numbers a and b increases, the number of operations in Euclid's algorithm grows quadratically. So if the size of the numbers doubles, the number of operations taken will go up by a factor of 4. This is quite a good algorithm. It is the same cost as using the highschool method for multiplying two integers -which is what Maple uses. For the interested reader who wants to know where the figure $O(n^2)$ for Euclid's algorithm comes from, we refer the reader to the non-trivial analysis in Knuth, *The Art of Computer Programming: Vol II Seminumerical Algorithms*. It turns out that the "worst case" of Euclid's algorithm is related to the Fibonacci numbers F_n . I.e. the numbers

```

> F := combinat[fibonacci]:

```

```
> seq( F(i), i=0..10 );
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

These numbers are a worst case because if we try to compute $\text{Gcd}(F(n), F(n-1))$ we will have to compute the $\text{Gcd}(F(n-1), F(n-2))$ and hence end up computing all the Fibonacci numbers in reverse order. Let's see that by tracing our latest version of GCD

```
> printlevel := 1000;
                                printlevel := 1000
```

```
> GCD( 55, 34 );
```

```
{--> enter GCD, args = 55, 34
```

```
    c := 55
```

```
    d := 34
```

```
    r := 21
```

```
    c := 34
```

```
    d := 21
```

```
    r := 13
```

```
    c := 21
```

```
    d := 13
```

```
    r := 8
```

```
    c := 13
```

```
    d := 8
```

```
    r := 5
```

```
    c := 8
```

```
    d := 5
```

```
    r := 3
```

```
    c := 5
```

```
    d := 3
```

```
    r := 2
```

```
    c := 3
```

```
    d := 2
```

```
    r := 1
```

```
    c := 2
```

```
    d := 1
```

```
    r := 0
```

```
    c := 1
```

```
    d := 0
```

```
    1
```

```
<-- exit GCD (now at top level) = 1}
```

```
    1
```

```
> printlevel := 1;
```

```
                                printlevel := 1
```

A natural question to ask is: can the Gcd of two integers of size $\leq n$ digits in length be computed in significantly fewer operations than $O(n^2)$? Can it be computed in say only $O(n)$ a linear number of operations. We can add two integers in $O(n)$ operations. What about Gcd's? Well, the answer is yes. We can compute the Gcd of two integers faster than $O(n^2)$ but not as fast as $O(n)$. See Knuth for a historical development. We want to conclude this worksheet by studying a different method that is particularly suited to computers. It is called the "Binary Gcd Algorithm". Like Euclid's algorithm, it is an $O(n^2)$ method, so as the size of the integers a and b increase, the cost of this method will grow quadratically. The advantage of this method is that if the numbers a and b are represented in a binary base $B = 2^m$ for some m , then certain operations can be done faster. The overall running time will be faster by a constant factor. If the implementation is good, perhaps by a factor of 2. The algorithm is like Euclid's algorithm: very simple. It relies on the same three observations of Euclid plus this one.

$$> \text{Gcd}(2*a, 2*b) = 2*\text{Gcd}(a, b);$$

$$\text{Gcd}(2a, 2b) = 2\text{Gcd}(a, b)$$

and the fact that the Gcd of two odd integers must be odd and the difference of two odd integers is even. The algorithm begins by writing $a = 2^i*a'$ and $b = 2^j*b'$ such that a' and b' are now odd. Hence $\text{Gcd}(a,b) = 2^{\min(i,j)} * \text{Gcd}(a',b')$. Determining i and j and dividing out by 2^i and 2^j is very easy if your integers are represented in a binary base. Here is the algorithm

```
> GCD := proc(a::nonnegint,b::nonnegint)
> local c,d,i,j;
> if b = 0 then
> a;
> elif a = 0 then
> b;
> elif irem(a, 2)=0 and irem(b, 2)=0 then
> 2*GCD(iquo(a, 2),iquo(b, 2));
> elif irem(a, 2)=0 then
> GCD(iquo(a, 2),b);
> elif irem(b,2)=0 then
> GCD(a, iquo(b, 2));
> elif a < b then
> GCD(b, a);
> else GCD(a-b, b)
> end if;
> end;
> GCD(3*64,3*4*5);
```

Now this routine is faster than the original version of Euclid's algorithm that we showed because in two steps it divides at least one of the numbers by 2. This is because if both a and b are odd, after doing a subtraction, it can divide by 2. Our implementation in Maple needs some improving. Here it is as an iterative procedure.

```

> GCD := proc(a::nonnegint,b::nonnegint)
> local c, d, i, j, t;
> if b = 0 then
> a;
> elif a = 0 then
> b;
> end if;
> c := a; d := b;
> for i from 0 while irem(c,2)=0 do
> c := iquo(c,2);
> end do;
> for j from 0 while irem(d,2)=0 do
> d := iquo(d,2);
> end do;
> if c < d then
> t := c; c := d; d := t;
> end if;
> do # Loop invariant c >= d and c,d are both odd
> if c = d then
> 2^min(i,j)*c ;
> end if;
> c := c-d;
> while irem(c,2)=0 do
> c := iquo(c,2);
> end do;
> if c < d then
> t := c; c := d; d := t;
> end if;
> end do;
> 2^min(i,j) * c;
> end:

```

The program has become trickier. And it is trickier to prove that it is correct. You'll notice that I've included a comment on the loop `do ... od`. The comment says that at the beginning of the loop the condition $c \geq d$ is supposed to always hold and both c, d are odd. This is called a loop invariant and such a thing is the key to proving a program with a loop correct. Can you argue that this program is really correct?

Finding out the cost of this program is much easier than Euclid's algorithm. Can you argue that the algorithm is $O(n^2)$?