Maple Mini-Course, Summer 2001

Jim Carlson

June 10, 2001

Contents

1 Introduction

Maple is a general-purpose "computer algebra" system. We will learn to use it by doing examples that are relevant to our summer course, mainly number theory. Let's begin with arithmetic:

```
> (2 + 3*4 - 1)^2;
                                169

> 5^30;
                                931322574615478515625

> 12345/321;
                                4115/106

> evalf(12345/321);
                                38.45794393

> trunc(12345/321);
                                38

> 12345 mod 321;
                                147

> 30!;
                                265252859812191058636308480000000

> evalf( % );
                                                33
                               .2652528598 10
```

Note that Maple tries to do exact arithmetic with integers (whole numbers) and rational numbers (fractions). Maple can deal with *very* big numbers. Note also the various ways of dealing with the quotient `12345/321`. The obvious expression gives the quotient as a fraction in lowest terms. And the expression `evalf(12345/321)` gives an approximate decimal expansion, while `trunc(12345/321)` gives the *integer quotient*. This is what you compute when doing long division. What about the remainder? It is given by `12345 mod 321`.

Consider also the strange expression `evalf( % )`. The "verb" `evalf` means "evaluate as a floating point number," and the symbol "`%`" stands for the value of previous computation. What we have done is to compute the very large number

$$30! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots 30$$

in scientific notation. Note also that *all* Maple expressions end with a semicolon. Grammar and punctuation are even more important in computer programming

than in English composition. Since cmpturs are not very imaginative, they cnnot understand thngs tht dsobey the rules.

Maple has many built-in functions. One of these is `sqrt`, which computes the square root of a number. Try the following:

```
> sqrt(49);
> sqrt(50);
```

You'll note a big difference in the output, since one number is a perfect square, the other is not. Now let's try something fun. We've listed some "myster Maple code" below. But try to guess what you will get before you try it, then think about it again after you have tried it. (Reverse-engineering is a time-honored way of learning everything from clock repair to computer programming).

```
> sqrt(2.0);
> sqrt( % );
> sqrt( % );
> sqrt( % );
> sqrt( % );
> sqrt( % );
> # etc.
```

The symbol "`#`" opens a *comment*: Maple does ignores the entire line from the first occurence of `#` onward.

Problems

1. Compute $2^{2^n} + 1$ for $n = 1...6$.
2. Compute $\sqrt{1009}$ in decimal form.
3. Try this:

   ```
   > A := (7 - 2*sqrt(5)) / (4 + 3*sqrt(5));
   > rationalize(A);
   > expand( % );
   ```

   Does the result of this computation express any general pattern?
4. How many ways are there of shuffling a deck of fifty two cards? Find the exact number, and find it also in scientific notation. If you could examine each one of the 52! possible decks in one microsecond, how long would it take to examine them?
5. Compute $1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7$ as a fraction in lowest terms. Then find the *complete* decimal expansion. If need be, use code like `Digits := 40` to improve the accuracy of your computations.

6. Find the complete decimal expansion of (a) 1/27, (b) 1/7, (c) 1/13.

   Can you make some general conclusions about the decimal expansions of rational numbers? Can you prove that these conclusions are true?

   — *Calculation = evidence. Proof = certainty + understanding.*

7. Consider the number with the repeating decimal expansion

$$0.\overline{12345678987654321}.$$

   Here $0.\overline{123} = 0.123123123...$, etc. Express the given number as a fraction in lowest terms.

   Can you make some general conclusions about numbers which have decimal expansions like this? Can you prove that these conclusions are true?

8. Investigate the quantity $2^n \bmod n$ for various n. Do you see any patterns? Make a conjecture! Can you prove that your conjecture is true?

   — *Conjecture: fancy name for educated guess.*

9. Find a number x such that

$$7x \bmod 11 = 1.$$

   That is, find a number such that when multiplied by 7 and divided by 11 gives a remainder of 1.

2 Loops

Now let's consider the following bit of code:

```
> sqrt(2.0);
> for i from 1 to 10 do sqrt( % ); od;
```

Before executing it, try to guess what it does.

OK, now you've run the code, and you see that we have done in two lines what the code below does in eleven:

```
> sqrt( 2.0 );
> sqrt( % );
> sqrt( % );
> sqrt( % );
> sqrt( % );
> sqrt( % );
> sqrt( % );
> sqrt( % );
> sqrt( % );
> sqrt( % );
> sqrt( % );
```

A construction like

```
for i from 1 to 10 do xxx; od;
```

where `xxx` is done over and over again, is called a *loop*. It is one of the most important computational tools. Our loop does its thing ten times, and uses the *loop counter* `i` to keep track of its progress.

Let's look at more examples of loops. The next one computes the sum of the first N integers.

```
> N := 10;
                          N := 10
> S := 0;
                          S := 0

> for i from 1 to N do S := S + i; od;

                          S := 1
                          S := 3
                          S := 6
                          S := 10
                          S := 15
                          S := 21
                          S := 28
                          S := 36
                          S := 45
                          S := 55
```

Of course this example is a little bit silly, since there is a formula for this kind of computation:

$$1 + 2 + 3 + \cdots + N = \frac{N(N+1)}{2}$$

Anyway, despite its silliness, there are lessons to be learned. First, notice that this time we have used *variables* in our loop. There are three of them, namely i, our loop counter; N, which is the number of terms; and S, which holds the sum which we build up step by step. We assign a value to a variable by an expression like

```
> S := 0;
```

In this case we set the value of S to zero.

Now let's try something less silly. We will try to find an approximate value of the sum

$$1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots$$

It is an infinite sum, so the best we can do add up the first N terms:

$$S_N = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{N^2}$$

The S_N's are called *partial sums*. To compute them we tinker with our previous example:

```
> N := 9; S := 0;
> for i from 1 to N do S := S + 1.0/i^2; od;
```

The sequence of partial sums S_1, S_2, S_3, etc. is

```
1.000000000,   1.250000000,   1.361111111,
1.423611111,   1.463611111,   1.491388889,
1.511797052,   1.527422052,   1.539767731, etc.
```

Is there a trend? Is there sum number towards which the S_N are getting closer and closer? If there is such a magic number, we call it the *limit value*, and we write

$$\lim_{N\to\infty} S_N = L.$$

What do you think is happening here? Is there really a limit value? If so, can you compute its value accurately to four decimal places? Can you *be sure* that your computation is correct?

Note. If you want to compute S_N for N very large, you may not want to display all the intermediate work. Here is a way of doing this:

```
N := 9; S := 0;
for i from 1 to N do S := S + 1.0/i^2; od:
S;
```

We put a colon a the end of the loop instead of a semicolon. This suppress the output. Then we put the statement "`S;`" after the loop. It evaluates and displays `S` so that we can see what the last value of `S` is. On the other hand, if you want more elaborate intermediate output, you can do this:

```
N := 9; S := 0;
for i from 1 to N do
   S := S + 1.0/i^2;
   print( i, S);
od;
```

Problems

1. You can use a loop to reconsider problem 8 of the previous section. The loop can generate more data if you need it:

```
N := 20;
for m from 2 to N do print( m, 2^m mod m); od;
```

What is the pattern?

2. You can generate data to solve equations like $14x = 3 \bmod 19$ by making part of a multiplication table for the numbers modulo 19:

```
N := 19; a := 14;
for k from 0 to N-1 do
   print( k, a*k mod N );
od;
```

What is the solution? Is there a better method for finding it?

3. Investigate the following sums

$$1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots$$

$$1 + \frac{1}{2^3} + \frac{1}{3^3} + \frac{1}{4^3} + \cdots$$

$$1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$$

4. Investigate also the sums

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots$$

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} \pm \cdots$$

For the second of these you will need to add some terms, subtract others. Here is a way to do this:

```
>  N := 1000;
   S := 0;
   for i from 1 to N do
      if type(i,odd)
         then S := S + 1.0/i;
         else S := S - 1.0/i;
      fi;
   od;
   S;
```

Note the use of the *conditional* statement,

```
if  X  then  P; else Q;   fi;
```

If `X` is true, then we do `P`. Otherwise we do `Q`. The `else` clause is optional.

Notes

Other than simple one-line expressions like `evalf( sqrt( 2 ) )`, you've learned only two programming structures: *loops* (the `for-do` statement), and *conditionals* (the `if-then-else` statement). But you can do a great deal of interesting and useful computing with these two structures and their close relatives.

3 Functions

We have seen that Maple has various built-in functions, like `sqrt`. We can also define our own functions, like the one below:

$$f(x) = \frac{1}{2}\left(\frac{3}{x} + x\right) \tag{1}$$

The "mystery function" is defined like this in Maple:

```
f := x -> (1/2)*(3/x + x);
```

Then we can compute its values, e.g., `f(2), f(11), f(2.34)`, etc. We can also compute expressions like `f( u + 1/v )`. And we can make a graph of f. Here is how we graph f on the interval $[0, 5] = \{\ x\ \mid 0 \le x \le 5\ \}$:

```
plot( f(x), x = 0..5 );
```

If you want to know more about plot, try

```
?plot
```

The graph of f is nice looking, but what is really interesting is to investigate the sequence

$$1.0,\ f(1.0),\ f(f(1.0)),\ f(f(f(1.0))),\ \ldots$$

If we compute by hand, we get

$$1.0,\ 2.0,\ 1.75,\ 1.732, \ldots,$$

which we could also write as

$$x_0,\ x_1,\ x_2,\ x_3,\ \ldots.$$

Note that the sequence satisfies the rule

$$x_{n+1} = f(x_n),$$

which just says "apply f to get the next term of the sequence from the previous one." Of course we have to have a place to start:

$$x_0 = 1.$$

To investigate the sequence experimentally by hand is tedious. A calculator helps, and a computer helps even more. Here is some Maple code which helps avoid the tedium:

```
x := 1.0;
for i from 1 to 10 do
    x := f(x);
od;
```

All we do is repeatedly replace `x` by `f(x)`. Try the code. How does the sequence $\{ x_n \}$ behave? Can you explain its behavior? Can you use this behavior to do something else?

Problems

1. Here is a way of getting an approximate root for an equation like

   $$x^3 + 2x = 1.$$

   First, note that the related equation $x^3 = 1$ has an exact root, $x = 1$. Next, rewrite the equation as

   $$x = \frac{1 - x^3}{2.0}$$

   and use this to generate a sequence with $x_0 = 1$, $x_1 = f(x_0)$, $x_2 = f(x_1)$, etc. The sequence begins like this:

   ```
   1.000000, 0.500000, 0.437500, ...
   ```

   In fact, the sequence converges to a limit L:

   $$\lim_{n \to \infty} x_n = L$$

   and the limit is a root of the equation. Use Maple to find a root accurate to six decimal places.

2. Find a root the equation $x^3 + 1.4x = 1$. Examine the related equations $x^3 + ax = 1$ for various a. Describe your findings.

3. Let $f(x) = (1/2)(5/x + x)$. Examine the sequence determined by the equations

   $$x_0 = 1$$

   and

   $$x_{n+1} = f(x_n)$$

   What can you say about the limit (if any) of this sequence?

4. A value $x = c$ is a *fixed point* of f if $f(c) = c$. Does the function defined in (1) have a fixed point? Does this explain anything?

5. Suppose that a sequence $\{ x_n \}$ is generated by the rule $x_{n+1} = f(x_n)$. Suppose that one encounters a repeated value in the sequence, say $x_M = x_N$. What happens?

6. Consider the function $f(x) = 7x \bmod 11$. Study the sequence defined by repeatedly applying f to a given initial value. Investigate the sequences obtained when the moduli are 12 and 13. Do you see any promising areas of investigation?

7. You have already solved equations like $7x = 1 \bmod 11$. (If you haven't go back and do this before going on!) Now suppose hat we want solve an equation of the form $f(x) = y$, where, for example,

$$f(x) = 531x \bmod 1001.$$

Sometimes (as is the case here), there are elegant and efficient methods, but for now we will consider a brute force method. The idea is that if there is a solution, it will be an integer in the range $0 \leq x \leq 1000$. (Why is this?) So we can just compute $f(0)$, $f(1)$, $f(2)$, etc., and stop when when we find an x such that $f(x) = y$. Here is code which does this:

```
isearch := proc(f,y,a,b)
     local x;
      x := a;
      while ( f(x) <> y ) and ( x <= b )
         do x := x + 1;
      od;
      if x <=  b then
         RETURN( x );
      else
         RETURN( 'NO SOLUTION' );
      fi;
end;
```

The name `isearch` was chosen to stand for "search an interval." We've built in an error message in case no solution is found. Now try this:

```
> f := u -> 531*u mod 1001;
> isearch(f, 1, 0, 1000);

> g := w -> 530*u mod 1000;
> isearch(g, 1, 0, 999);
```

(a) Could you have predicted the result for $g(x) = 1$ in advance? (b) Use `isearch` to solve the equation $531x = 1 \bmod 1009$.

Notes

The `while - do` loop you saw in the last problem is different from the `for - do` loop: you can't tell by looking at the loop how many times the "loop body" will be executed. (The body is the part between `do` and `od`).

4 Primes

Let's use what we have learned so far to design a function which factors an integer into primes. The idea for factoring an integer n is

> ($*$) Set a "trial divisor" d to 2. If d divides n, write down d, then replace n by n/d. Otherwise increase d by 1. If $n = 1$, stop. Otherwise go to ($*$).

Here is a function, `nfactor`, which does the job. First, an example:

```
> nfactor( 1234 );
                          [2, 617]

> nfactor( 12345689 );
                          [17, 751, 967]
```

You can easily check that the factors are correct. Note the asymmetry of the factoring problem: checking factors is easy, but finding them is hard.

The heart of `nfactor` is the code below, which is a new kind of loop:

```
while m > 1 do
  if m mod d = 0
  then
    print( d );
    m := m/d;
  else d := d + 1;
  fi;
od;
```

The part between `while` and `od` is the *loop body*. It is executed (run) as long as the condition $n > 1$ holds. In other words, the loop runs as long as there are factors to be found. The loop sits inside a definition of `nfactor` which is a function of the variable `n`. The structure of such a definition is

```
< function-name > = proc( < list of variables > )
       < definition >
       end;
```

Putting the loop inside the procedure definition and adding a few other important ingredients, we get the working code:

```
nfactor := proc( n )
   local d, m;
   m := n;
   d := 2;

   while m > 1 do
     if m mod d = 0
     then
       print( d );
       m := m/d;
     else d := d + 1;
     fi;

   od;
end;
```

The variables `d, n` are *local*, and cannot be used or seen outside the procedure definition.

Our factoring program can be improved in many ways. One way is to accumulate the factors in a list `L`, then return the list as the value of `nfactor`. This is the version of the factoring program used in the examples above: a typical list is `[17, 751, 967]`.

Here is what you need to start using lists. First, we define a list by statements like the ones below:

```
A := [];
B := [1, 2, 3, 4];
C := [17, 751, 967];
```

The first if these lists is the *empty list*. Second, we can access list elements by expressions like

```
C[1];
```

In this case, `C[1] = 17`. For the number of elements in a list, or "number of operands," use `nops(L)`. To concatenate two lists, use an expression like

```
[ op(B}, op(C) ];
```

All `op` does is to strip away the brackets on both sides. Note what happens if you don't do this:

```
> [ A, B ];
                [[1,2,3,4],[17, 751, 967]];
```

What do you think the value of `[ op(A), op(B) ]` is? Now we can modify our original version of `nfactor` to produce the improved one:

```
nfactor := proc( n )
   local d, m, F;
   m := n;
   d := 2;
   F := [];

   while m > 1 do
     if m mod d = 0
     then
       F := [ op(F), d ];
       m := m/d;
     else d := d + 1;
     fi;
   od;

RETURN( F );
end;
```

Notice the expression `RETURN( F )`. It determines what value is returned when we say something like `nfactor(1234)`. We could say something else, e.g., `RETURN( m )`, but this would be stupid.

Problems

1. Factor the following numbers: 1234, 1235, 1236, ... 1244. Any observations or questions for investigation?

2. Factor the following numbers: 12341234, 12341235, 12341236, ... 12341244. Any observations or questions for investigation?

3. Consider numbers of the the form $2^n - 1$. Any ideas about when these might be prime?

4. Let's consider the factorization of random numbers. The easiest way to do this is to use Maple's `rand` function:

```
for i from 1 to 10 do
    rand();
 od;
```

The result of this bit of code is 10 random numbers. To factor ten random numbers, try this:

```
for i from 1 to 10 do
    nfactor( rand() );
 od;
```

Any observations? Perhaps more experimentation and thought is necessary?

5. To study random numbers in a specified range, say 1000 to 2000, do this:

```
RN := rand(1000..2000);
for i from 1 to 10 do
    nfactor( RN() );
 od;
```

Any observations? Perhaps more experimentation is necessary?

Note: `rand` is a function of *zero* arguments, just as `sqrt` is a function of one argument. That is why we must write `rand()`, just as we may write `sqrt(49)`, `sqrt(11.0)`, etc.

6. Consider the numbers
$$F_n = 2^{2^n} + 1$$
On the basis of the evidence for $n = 1,\ 2,\ 3,\ 4$, the French mathematician Pierre Fermat (1601–1665) conjectured that these numbers are prime. What can you say about Fermat's problem?

7. Can `nfactor` be improved (speeded up)?

Notes

1. http://www.utm.edu/research/primes/

5 Diophantine Equations I

A Diophantine equation is an equation like
$$x^2 + y^2 = z^2$$

with integer equations for which we seek integer (or sometimes rational) solutions. Integer solutions to this equation are *Pythagorean triplets*: they represent right triangles with integer-length sides.

Now it is easy to check whether a proposed solution of a Diophantine equation really is one. For example, the vector $(x, y, z) = (3, 4, 5)$ is a solution, but $(x, y, z) = (3, 4, 6)$ is not. Consequently we can use trial and error to search for solutions. Such a "brute force attack" is tedious and slow, but can be speeded up enormously with a computer. We ilustrate this with the `ptriple search` code below. It finds all solutions satsifying the bounds $a \leq x \leq y \leq z \leq b$. For example, if we say `ptriplesearch(2,20)`, we get the output

```
 3,  4,  5
 5, 12,  13
 6,  8,  10
 8, 15,  17
 9, 12,  15
12, 16,  20
     6
```

Thus there are just six triplets in the given range. Here is the code:

```
ptriplesearch := proc(a,b)
  local i,j,k,count;
    count := 0;
    for i from a to b do
      for j from i to b do
        for k from j to b do
          if  i^2 + j^2 = k^2
          then
            print( i, j, k );
            count := count + 1;
          fi;
        od;
      od;
    od;
  RETURN( count );
end;
```

Note that in the output of `ptriplesearch(2,20)`, some of the triplets are multiples of others. For example, (6,8,10) is twice the vector (3,4,5). We can exclude these by not recording triples (x, y, z) such that x and y have a common factor. (Why?) The `gcd` function computes the greatest common divisor of two numbers, e.g., `gcd(20,15) = 5`. To exclude the uninteresting solutions, we change the line

```
if  i^2 + j^2 = k^2
```

to

```
if  i^2 + j^2 = k^2 and gcd(i,j) = 1
```

Note that the `gcd` function is part of Maple. In the next section we will see how to design such a function from scratch.

Note. Pythagorean triples like (3,4,5) that have no common factors are called *primitive*.

Brute force attack

The kind of brute force attack that we used to find Pythagorean triplets can be mounted against any Diophantine equation. Consider, for example, the cubic equation

$$y^2 = x^3 + 17;$$

We can look for solutions in the box $0 \leq x \leq a$, $0 \leq x \leq b$ by using the `boxsearch(f,a,b)` code listed below, where

```
f := (x,y) -> y^2 - x^3 - 17;
```

For example, to find solutions in the box $0 \leq x \leq 100$, $0 \leq y \leq 100$, we run the code

```
boxsearch(f,100,100);
```

The solutions in this range are $(2, 5)$, $(4, 9)$, and $(8, 23)$. Anyway, here is the code:

```
boxsearch := proc(f,a,b)
   local i,j;
   for i from 0 to a do
     for j from 0 to b do
       if f(i,j) = 0 then print((i,j)); fi;
     od;
   od:
end:
```

Problems

1. Modify `ptriplesearch` as suggested above, and use it to list all primitive Pythagorean triples satisfying $1 \leq x \leq y \leq z \leq 100$. How many are there. Do you noitice any patterns?

2. It is clear that a Pythagorean triple with $x \leq y \leq z$ must actually satisfy $y < z$. In fact, $x < y$ as well. Try to prove both facts.

3. In view of the previous problem, we ought modify the `ptriplesearch` code to inspect only triplets such that $x < y < z$. Here is the needed change:

```
for i from a to b do
  for j from i+1 to b do
    for k from j+1 to b do
```

   Now let's do some counting: (a) how many integer vectors satisfy $1 \leq x < y < z \leq N$? (b) let $\delta(N)$ be the density of primitive Pythagorean triples in the given search space: the ratio of the number of such triples satisfying the inequality to the number of all integer vectors satisfying the inequality. Compute $\delta(100)$. Do the odds favor finding Pythagorean triplets?

4. Its time to talk about running time. We run the following experiment to compute the number of seconds needed to execute the `ptriplesearch` code for inputs of various sizes:

```
> time( ptriplesearch(1,100) );
> time( ptriplesearch(1,200) );
> time( ptriplesearch(1,400) );
```

   On my computer I found the following:

```
-------------
   N       T
-------------
 100       4
 200      35
 400     299
-------------
```

   Note that when we double N, we multiply the running time by a factor of roughly eight. To be more precise, we find $\log_2(35/4) \approx 3.13$, $\log_2(299/32) \approx 3.09$, for an average value of about 3.11. Thus

$$T(N) \approx 4.0 \times \left(\frac{N}{100}\right)^{3.11} .$$

   In other words, if we double N we multiply $T(N)$ by a factor of $2^{3.11} \approx 8.63$. Use this empirical scaling law to determine how long it will take to execute `ptriplesearch(1,N)` for $N = 1000$, $N = 10,000$, and $N = 100,000$. Commment on your results.

5. Find running times for `ptriplesearch(1,N)` on your computer, and find the empirical relation between T and N.

6. We've just spent some effort finding an empirical relation between T and N. Let's go back and see what we can accomplish with pure thought. We'll do a back of the envelope computation. The vectors to be inspected lie in an $N \times N \times N$ cube. Moreover, the number of integer vectors in the cube $1 \leq x \leq N$, $1 \leq y \leq N$, $1 \leq z \leq N$ is exactly N^3. The number of integer vectors (aka lattice points) satisfying $1 \leq x < y < z \leq N$ is roughly some fixed fraction of the cube. Thus the number of lattice points to be inspected is roughly cN^3. Therefore if we double N, we multiply the running time by a factor of eight.

   Our "back-of-the-envelope" calculation is pretty good agreement with our empirically derived scaling law. Now find the constant c.

7. Find all integer solutions of $y^2 = x^3 + 17$ in the box $0 \leq x \leq 1000$, $0 \leq y \leq 1000$.

   Research question: Can we determine *all* integer solutions of the given equation?

8. Does the equation $y^2 = x^3 + 17$ have rational solutions (fractions) which are not integers? If there are such solutions, are they rare or abundant?

9. What does the set of real solutions to $y^2 = x^3 + 17$ look like? You should first try to work the general shape of the solution set by hand. Then experiment with code like

```
> with( plots );   # load plots package
> L := 10;         # define a box
> implicitplot( y^2 - x^3 - 17, x = -L..L, y = -L..L );
```

10. Find a complex solution of $y^2 = x^3 + 17$ which is not real.

    Research question: what does the set of complex solutions look like?

6 Diophantine Equations II

We are now going to show how to compute the greatest common divisor (gcd) of two numbers using what is called *recursion*, a very powerful programming tool. The goal is a function *gcd* which such that $gcd(6, 15) = 3$, $gcd(7, 11) = 1$, etc. As a bonus, we we learn how to *very* efficiently solve linear Diophantine equations, that is, equations of the form

$$ax + by = c.$$

Before we can design *gcd*, we must understand the relevant mathematics. The first observation is that given any two positive integers, we can divide one

by the other, producing an integer quotient and remainder. This is what long division does for us. For example, the quotient of 12345 by 321 is 38, and the remainder is 147. In general, if we divide a by b, we obtain a quotient q and a remainder r, where

$$a = bq + r, \qquad 0 \leq r < b$$

This is what is called the *division algorithm*. Now suppose that d divides both a and b. That is, d is a common divisor. Since $r = a - bq$, d also divides r: it divides the right-hand side of the previous equation and so also divides the left-hand side. Therefore divisors of a and b are divisors of b and r. Is the reverse true? Well, if d divides b and r, then it also divides the right-hand side of $a = bq + r$, and so it divides the left-hand side. We have arrived at the following important result:

$$gcd(a, b) = gcd(b, r).$$

We need just one more observation. Suppose b divides a. Then what is the greatest common divisor? Well, this is clear:

$$\text{if } b \text{ divides } a \text{ then } gcd(a, b) = b.$$

Now we can design our gcd function. We'll call it `xgcd` since it is experimental:

```
xgcd := proc(a,b)
   local r;
   r := a mod b;
   # print(a.b,r);
   if r = 0
     then RETURN(b);
     else xgcd(b,r);
   fi;
end;
```

Our definition is recursive because it "curves back on itself:" the definition of `xgcd` refers to `xgcd`. Notice how we have faithfully translated our two mathematical properties of *gcd* into bits of code. This is generally the way it is with recursively defined functions.

Linear Diophantine Equations

Let us now take up the problem of solving Diophantine equations of the form

$$ax + by = c$$

These *linear* equations are completely understood, and there is a very efficient method for finding solutions which works whenever the gcd of a and b divides c. It is an elaboration of the Euclidean algorithm, and operates as follows. Either

b divides a or it does not. In the first case, b is the gcd of a and b, so it divides c. Then $(x, y) = (0, c/b)$ is a solution. In the second case, write

$$a = bq + r$$

using the division algorithm, and then substitute into the given equation to obtain

$$(bq + r)x + by = c.$$

Rearrange as

$$b(qx + y) + rx = c.$$

Introduce a new variable by

$$u = qx + y,$$

and subsitute this into the previous equation to obtain

$$bu + rx = c$$

Now the gcd of b and r is the same as that of a and b, so the last equation is solvable. We use a solution of this "smaller" equation to solve the original one using the substitution of variables. And now we embody these ideas in the Maple code below. You should check that the two parts of the `then`–`else` clause correspond to the two mathematical cases just considered.

```
xsolve := proc(a,b,c)
             local r, q, s, u, x;
             r := a mod b;
             if r = 0
                then RETURN( [0,c/b] );
                else
                   s := xsolve( b, r, c );
                   u := s[1]; x := s[2];
                   q := (a-r)/b;
                   RETURN( [ x, u - q*x ] );
             fi;
          end:
```

Problems

1. Find the gcd of 119521589474 and 574157513825. Check your results with `nfactor`.

2. Run a contest between `nfactor`, `xgcd`, and `gcd` for finding the gcd of he numbers in the previous problem.

3. Repeat the previous problem with 100119521589474 and 100574157513825.

4. Solve $113x + 301y = 1$. Check your solution!

5. Solve $150650905007x + 553408748182y = 1$. Check your solution! Then run a contest between `xsolve` and `boxsearch`.

   Note once again that checking that solutions are indeed solutions easy, while finding solutions is hard.