MAPLE Notes for MACM 442 / MATH 800 / CMPT 881

Michael Monagan
Department of Mathematics
Simon Fraser University
August, 1998.
Updated August 2002, September 2004, September 2006.

```
> restart;
```

These notes are for Maple V Release 10. They are platform independent, i.e., they are the same for the Macintosh, PC, and Unix versions of Maple. These notes should be backwards compatible with Maple 8 and Maple 9.

Maple as a Calculator

Input of a numerical calculation uses +, -, *, /, and ^ for addition, subtraction, multiplication, division, and exponentiation respectively.

```
> 1+2*3-2;
```

$$5$$

```
> 2^3;
```

$$8$$

```
> 120/105;
```

$$\frac{8}{7}$$

Because the input involved integers, not decimal numbers, Maple calculates the exact fraction when there is a division, automatically cancelling out the greatest common divisor (GCD). In this case the GCD is 15, which you can calculate specifically as

```
> igcd(120,105);
```

$$15$$

Observe that every command ends with a semicolon ; This is a gramatical requirement of Maple. If you forget, Maple will assume that the comand is not complete. This allows you to break long commands across a line. For example

```
> 1+2*3/
> (2+3);
```

$$\frac{11}{5}$$

For some analysis with probabilities, we will need to use decimal arithmetic. The presence of a decimal point . in a number means that the number is a decimal number and Maple will, by default, do all calculations to 10 decimal places.

```
> 120/105.0;
```

$$1.142857143$$

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

$$1.414213562$$

```
> sqrt(2);
```

$$\sqrt{2}$$

Notice the difference caused by the presence of a decimal point in these examples. Now, if you have input an exact quantity, like the $\sqrt{2}$ above, and you now want to get a numerical value to 3 decimal digits, use the evalf command to evaluate to floating point. Use the % character to refer to the previous Maple output.

```
> evalf(%,3);
```

$$1.41$$

To input a formula, just use a symbol, e.g. x and the arithmetic operators and functions known to Maple. For example, here is a quartic polynomial in x .

```
> x^4-3*x+2;
```

$$x^4 - 3x + 2$$

We are going to use this polynomial for a few calculations. We want to give it the name f so we can refer to it later. We do this using the assignment operation in Maple as follows

```
> f := x^4-3*x+2;
```

$$f := x^4 - 3x + 2$$

The name f is now a variable. It refers to the polynomial. Here is it's value

```
> f;
```

$$x^4 - 3x + 2$$

To evaluate this as a function at the point $x = 3$ use the eval command as follows

```
> eval(f,x=3);
```

$$74$$

The following commands differentiate f with respect to x and factor f into irreducible factors over the field of rational numbers.

```
> diff(f,x);
```

$$4x^3 - 3$$

```
> factor(f);
```

$$(x-1)(x^3 + x^2 + x - 2)$$

You can graph functions using the plotting commands. The basic syntax for the **plot** command for a function of one variable is illustrated as follows:

```
> plot(f,x=0.5..1.5);
```

In the graph I can see a local minimum near x=0.9. We can find this point using calculus. The command **fsolve**(f(x)=0, x), on input of a polynomial *f(x)* computes 10 digit numerical approximations for the real roots of *f(x)*.

```
> fsolve(diff(f,x)=0,x);
```

$$0.9085602964$$

We have used the name f as variable to refer to formulae and the symbols x for an uknown in a formula. Often you will have assigned to a name like we have done here to f but you want now to use the name f as a symbol again, not as a variable. You can unassign the value of a name as follows

```
> f;
```

$$x^4 - 3x + 2$$

```
> f := 'f';
```

$$f := f$$

```
> f;
```

$$f$$

To find out how to use Maple commands, such as plot, factor, eval, and diff, go to the Maple help page by typing ?plot
and look at the examples. Do this now if you are using Maple for the first time. You will get a "help window".

```
> ?eval
> ?plot
> ?diff
> ?fsolve
>
```

Integers

Here are some commands which do integer calculations that we will use in the course. Integers in Maple are not limited to the precision of the hardware on your computer. They are limited by an internal limit of Maple to approximately half a million digits. Any calculations that you do with large integers though will take longer for larger integers. Here is 2^{100}.

```
> 2^100;
```

$$1267650600228229401496703205376$$

The command irem(a,b) computes the integer remainder of a divided by b. The command iquo(a,b) computes the integer quotient of a divided by b. For example

```
> a := 20;
  b := 7;
```

$$a := 20$$

$$b := 7$$

```
> r := irem(a,b);
```

$$r := 6$$

```
> q := iquo(a,b);
```

$$q := 2$$

It should be the case that $a = b\,q + r$. Let's check

```
> a = b*q + r;
```

$$20 = 20$$

The commands igcd(a,b) and ilcm(a,b) compute the greatest common divisor and least common multiple of integers a and b, respectively.

```
> igcd(6,4);
```

$$2$$

```
> ilcm(6,4);
```

$$12$$

The command **igcdex**(*a,b,'s','t'*) outputs $g = \mathrm{GCD}(a, b)$. It also assigns s, t integers satisfying the equation $s\,a + t\,b = g$ and satisfying $|s| < |b|$ and $|t| < |a|$. So this command implements the extended Euclidean algorithm. For example

```
> g := igcdex(3,5,'s','t');
```

$$g := 1$$

```
> s;
```

$$2$$

```
> t;
```

$$-1$$

```
> s*3+t*5;
```

$$1$$

The command **isprime**(*n*) outputs false if *n* is composite and true if *n* is prime.
The command **ifactor**(*n*) computes the integer factorization of an integer.

```
> isprime(997);
```

$$true$$

```
> isprime(1001);
```

$$false$$

```
> ifactor(1001);
```

$$(7)\ (11)\ (13)$$

For the RSA cryptosystem we need large primes, 100 digit primes. The command **nextprime**(n) outputs the first prime greater than or equal to n and the command **prevprime**(n) finds the first prime less than or equal to n. Here is the first 100 digit prime.

```
> prevprime(10^100);
```

```
99999999999999999999999999999999999999999999999999999999999999999999999999999\
    99999999999999999999203
```

Several of the cryptosystems require that we generate random primes, or primes with certain properties. This needs to be done with some care. For example, the prime we just generated is not a good prime simply because of the way we created it. We need a random prime. Note, Maple's random number generator rand(...) is a "pseudo-random-number-generator". It will always create the same sequence of random numbers.

```
> R := rand(0..10^10-1);
```

$$R := \mathbf{proc}() (\mathbf{proc}() \mathbf{option}\ builtin;\ 391\ \mathbf{end\ proc})(6, 10000000000, 34)\ \mathbf{end\ proc}$$

So R is the random number generator. It's a Maple procedure. You call it with no arguments to get random numbers.

```
> R();
```

$$581869302$$

```
> R();
```

$$4161255391$$

For a list of integer commands see

```
> ?integer
>
```

Lists

The simplest data structure in Maple is a list. The elements of a list may be of any type. To create a list of values enclose them in square brackets [,]. Lists may be nested of course and the entries may be of any type.

```
> restart;

> E := []; # the empty list
```

$$E := [\,]$$

```
> L := [1,2,-3,4,1];
```

$$L := [1, 2, -3, 4, 1]$$

```
> M := [[1,2,3],[x,y,z]];
```

$$M := [[1, 2, 3], [x, y, z]]$$

To count the number of entries in a list use nops(L) command.

```
> nops(L);
```

$$5$$

```
> nops(M);
```

$$2$$

To access the i'th element of a list (counting from 1) use a subscript. A negative subscript counts from the end.

```
> L[3];
```

$$-3$$

```
> L[-1];
```

$$1$$

```
> M[2];
```

$$[x, y, z]$$

```
> M[2][2];
```

$$y$$

Use the following to extract a sublist

```
> L[2..3];
```

$$[2, -3]$$

```
> L[2..-1];
```

$$[2, -3, 4, 1]$$

To append (prepend) elements to a list use the following.

```
> op(L);
```

$$1, 2, -3, 4, 1$$

```
> [op(L),5];
```

$$[1, 2, -3, 4, 1, 5]$$

To test if an element is in a list use

```
> member(2,L);
```

$$true$$

```
>
```

Loops

```
> restart;
```

To do a sequence of calculations it will be handy to know how to use some of Maple's looping commands and also the if command. To execute a command in Maple conditionally use the if command which has either of the following forms

if <condition> **then** <statements> **else** <statements> **fi**

or just

if <condition> **then** <statements> **fi**

```
> if 2>1 then print(good) else print(bad) fi;
```

$$good$$

To execute one or more statements zero or more times in a loop use the for command. It has the following form

for <variable> **from** <start> **to** <end> **do** <statements> **od**

```
> for i from 1 to 5 do i^2; od;
```

$$1$$
$$4$$
$$9$$
$$16$$
$$25$$

To execute some statements while a condition is true use the while loop. It has the syntax

while <condition> **do** <statements> **od**

In the following example we repeatedly divide an integer n by 2 until it is odd.

```
> n := 12; while irem(n,2) = 0 do n := iquo(n,2); od;
```

$$n := 12$$
$$n := 6$$
$$n := 3$$

Here is a loop to calculate the GCD of two integers *a* and *b* using Euclid's algorithm. Notice that this loop has three statements in the body of the loop - between the **do** ... **od**, each of which is terminated by a

semicolon. You don't have to put them on the same line as I have done here.

```
> a := 64; b := 20;
```

$$a := 64$$

$$b := 20$$

```
> while b <> 0 do
>    r := irem(a,b); a := b; b := r;
> od;
```

$$r := 4$$

$$a := 20$$

$$b := 4$$

$$r := 0$$

$$a := 4$$

$$b := 0$$

Thus 4 should be the GCD(64,20). A check with Maple

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

$$4$$

Two other useful looping constructs are the **map** command and the **seq** command. The examples show what the commands do.

```
> L := [1,2,3,4,5];
```

$$L := [1, 2, 3, 4, 5]$$

```
> map( f, L );
```

$$[f(1), f(2), f(3), f(4), f(5)]$$

```
> map( isprime, L );
```

$$[false, true, true, false, true]$$

```
> seq( f(n), n=L );
```

$$f(1), f(2), f(3), f(4), f(5)$$

```
> seq( isprime(n), n=L );
```

$$false, true, true, false, true$$

```
> L := [seq( n^2, n=L )];
```

$$L := [1, 4, 9, 16, 25]$$

Read the help files for these commands, they are very handy.

```
> ?map
> ?seq
>
```

Modular Arithmetic

Modular arithmetic is done using the mod operator in Maple. By default, Maple uses the positive range for the integers modulo *m*, that is, the result is calculated in the range 0 .. *m* – 1.

```
> restart;
> 12 mod 7;
```

$$5$$

```
> 2+3*3 mod 7;
```

$$4$$

To compute $a^{(-1)} \bmod m$, you can do either of the following

```
> 2^(-1) mod 7;
```

$$4$$

```
> 1/2 mod 7;
```

$$4$$

To compute a^n mod m you can do either

```
> 2 ^ 200 mod 7;
```

$$4$$

```
> 2 &^ 200 mod 7;
```

$$4$$

Use the latter. The difference is that in the first case, the integer 2^{200} was computed then reduced modulo m. In the second case, all products were reduced modulo m so no large integers occured.

We will use a loop to verify that Fermat's (little) theorem holds for p = 7.

```
> p := 7;
  for i from 0 to p-1 do (i^p mod p) = i od;
```

$$p := 7$$
$$0 = 0$$
$$1 = 1$$
$$2 = 2$$
$$3 = 3$$
$$4 = 4$$
$$5 = 5$$
$$6 = 6$$

We can solve equations and systems of equations modulo n using the **msolve** command.

```
> msolve( 6*x=4, 13 );
```

$$\{x = 5\}$$

```
> msolve( 6*x=4, 26 );
```

$$\{x = 5\}, \{x = 18\}$$

```
> msolve( {24*a+b=5, 4*a+b=9, 18*a+b=1}, 26 );
```

$$\{a = 18 + 13\ \_Z2\sim,\ b = 15\}$$

The variable _Z2~ means any integer so the solutions are {b = 15, a = 18} and {b = 15, a = 5}.

```
>
```

Number Theory

Some relevant integer functions are available in the number theory package.
This shows how to load a package.

```
> restart;
  with(numtheory);
Warning, the protected name order has been redefined and unprotected
```

[*GIgcd, bigomega, cfrac, cfracpol, cyclotomic, divisors, factorEQ, factorset, fermat, imagunit, index, integral_basis, invcfrac, invphi, issqrfree, jacobi, kronecker,* λ*, legendre, mcombine, mersenne, migcdex, minkowski, mipolys, mlog, mobius, mroot, msqrt, nearestp, nthconver, nthdenom, nthnumer, nthpow, order, pdexpand,* ϕ*,* π*, pprimroot, primroot, quadres, rootsunity, safeprime,* σ*, sq2factor, sum2sqr,* τ*, thue*]

The command **phi**(n) computes $\phi(n)$, the number of integers between 1 and n relatively prime to n.
Note this command is expensive because it factors n.

```
> phi(15);
```

$$8$$

The command **mcombine**(m1,u1,m2,u2) does Chinese remaindering to calculate the integer u satisfying $u = u_i \bmod m_i$, for $i = 1 .. 2$. For example

```
> u := mcombine(5,4,7,3);
```

$u := 24$

```
> u mod 5;
```

4

```
> u mod 7;
```

3

The command **msqrt**(x,n) computes a square root of *x* modulo *n* if it exists and outputs FAIL otherwise. Note this command factors *n* so this is also expensive if *n* is not prime.

```
> p := 11;
```

$p := 11$

```
> msqrt(2,p);
```

FAIL

```
> msqrt(3,p);
```

5

```
> 5^2 mod p;
```

3

The command **mlog**(x,m,n) computes the discrete logarithm of *x* base *m* modulo *n*, i.e. finds *y* such that $y^m \bmod n = x$. Note this command is expensive. Like integer factorization, no polynomial time algorithm is known for the discrete logarithm problem.

```
> x := 2 &^ 5 mod p;
```

$x := 10$

```
> mlog(x,2,p);
```

5

```
>
```

Strings

A string is input as text inside " (string) quotes. For example

```
> restart;
  A := "hello";
  B := "there";
```

$A :=$ "hello"

$B :=$ "there"

The number of characters in a string is given by **length**(s).

```
> length(A);
```

5

The empty string is "". To access the i'th character from a string use A[i], e.g.

```
> A[1]; A[2];
```

"h"

"e"

Negative subscripts count from the end of the string where position -1 refers to the last character in the string.

```
> A[-1]; A[-2];
```

"o"

"l"

If A is a string the notation A[i..j] selects the substring of characters from position i to j from A.

```
> A[1..2]; A[-2..-1];
```

"he"

"lo"

The command **cat**(s1,s2,...,sn) joins n strings together.

```
> cat(A," ",B);
```

"hello there"

The **seq** command can be used to extract the sequence of characters from a string and we can put them back together using the **cat** command.

```
> C := seq( A[i], i=1..length(A) );
```

C := "h", "e", "l", "l", "o"

```
> cat( C );
```

"hello"

Here is how you would extract all trigrams from a string.

```
> seq( A[i..i+2], i=1..length(A)-2 );
```

"hel", "ell", "llo"

There is more if you need it. See

```
> ?string
> ?StringTools
> with(StringTools):
Warning, the assigned name Group now has a global binding
> A;
```

"hello"

```
> Search("l",A);
```

3

```
> SearchAll("l",A);
```

3, 4

Hence the frequency of the letter "l" in the string A is

```
> nops([SearchAll("l",A)]);
```

2

```
>
```

Maple Functions and Procedures

A simple function, like the function ek(x) = a*x+b mod n may be input using the arrow notation in Maple, as follows

```
> ek := x -> 3*x+5 mod 26;
```

$ek := x \rightarrow (3\,x + 5) \bmod 26$

```
> ek(1);  ek(7);
```

8

0

We create the inverse function

```
> 1/3*(y-5) mod 26;
```

$9\,y + 7$

```
> dk := y -> 9*y+7 mod 26;
```

$dk := y \rightarrow (9\,y + 7) \bmod 26$

```
> dk(8);  dk(0);
```

1

7

So if we have encoded some text "BUYIBM" as a list of integers [1,20,24,8,1,12] we can encrypt it and decrypt it as follows.

```
> plaintext := [1,20,24,8,1,12];
```

$plaintext$:= [1, 20, 24, 8, 1, 12]

```
> ciphertext := map(ek,plaintext);
```

ciphertext := [8, 13, 25, 3, 8, 15]

```
> map(dk,ciphertext);
```

[1, 20, 24, 8, 1, 12]

For the Hill cipher, we which takes two inputs x1 and x2 and outputs two outputs y1 and y2 we will use a Maple procedure rather than the arrow functions.

A procedure in Maple takes the form

proc(p1, p2, ...)
local l1, l2, ... ;
global g, g2, ... ;
statement1;
statement2;
....
statementn;
end proc

There may be zero or more parameters, one or more locals, one or more globals and one or more statements in the procedure body.

The local and global statements are optional. Variables in the procedure body that are not explicitly declared as parameters, locals, or globals are declared to be local automatically if assigned to, otherwise they are global. The value returned by the procedure is the value of *statementn*, the last statement in the body of the procedure or the value of an explicit return statement. Type declarations for parameters and local variables need not be explicitly given. Some examples will help.

```
> f := proc(x) y := x^2; y-1; end proc;
Warning, ‘y‘ is implicitly declared local to procedure ‘f‘
```

f := **proc**(*x*) **local** *y*; *y* := *x*^2; *y* – 1 **end proc**

```
> f(2);
```

3

```
> f(z);
```

$z^2 - 1$

This next example searches a string s for the letter x. It outputs the position of the first occurrence of x in s and 0 otherwise. I am also telling Maple that the inputs should be of type string. Below is an example with inputs of type integer. See **?type** for how to specify types and for what types are available if you need them.

```
> position := proc(x::string,s::string) local i;
    for i from 1 to length(s) do if s[i]=x then return i fi; od;
    0; # meaning x is not in the list
  end proc;
```

position :=

proc(*x*::*string*, *s*::*string*) **local** *i*; **for** *i* **to** length(*s*) **do if** *s*[*i*] = *x* **then return** *i* **end if end do**; 0 **end proc**

```
> position("U","BUYIBM");
```

2

```
> position("V","BUYIBM");
```

0

```
> position(5,"BUYIBM");
Error, (in position) invalid input: position expects its 1st argument, x, to be of
type string, but received 5
```

Okay, now let’s code the Hill cipher. It is a function of two arguments x1 and x2 (integers) and it outputs two values y1 and y2.

```
> e := proc(x1,x2) local y1,y2;
      y1 := 11*x1+3*x2 mod 26;
      y2 :=  8*x1+7*x2 mod 26;
      (y1,y2);
  end proc:
```

```
> x := plaintext;
```

$$x := [1, 20, 24, 8, 1, 12]$$

```
> e(x[1],x[2]);
```

$$19, 18$$

```
> ciphertext := [seq( e(x[2*i+1],x[2*i+2]), i=0..iquo(nops(x),2)-1 )];
```

$$ciphertext := [19, 18, 2, 14, 21, 14]$$

Now to decrypt we need the inverse of this function. We compute the inverse of the corresponding matrix mod 26

```
> A := Matrix([[11,3],[8,7]]);
```

$$A := \begin{bmatrix} 11 & 3 \\ 8 & 7 \end{bmatrix}$$

```
> Inverse(A) mod 26;
```

$$\begin{bmatrix} 7 & 23 \\ 18 & 11 \end{bmatrix}$$

For the decryption function I'll use Maple's -> notation instead of a procedure. Let's check that d is the inverse of e then decrypt the ciphertext.

```
> d := (y1,y2) -> (7*y1+23*y2 mod 26, 18*y1+11*y2 mod 26);
```

$$d := (y1, y2) \rightarrow ((7\ y1 + 23\ y2) \bmod 26, (18\ y1 + 11\ y2) \bmod 26)$$

```
> d(e(x1,x2));
```

$$x1, x2$$

```
> e(d(x1,x2));
```

$$x1, x2$$

```
> y := ciphertext;
```

$$y := [19, 18, 2, 14, 21, 14]$$

```
> [seq( d(y[2*i+1],y[2*i+2]), i=0..2 )];
```

$$[1, 20, 24, 8, 1, 12]$$

This last example is an implementation of the Euclidean algorithm.

```
> EuclideanAlgorithm := proc(a::integer,b::integer) local c,d,r;
      (c,d) := (abs(a),abs(b));
      while d <> 0 do r := irem(c,d); (c,d) := (d,r); od;
      c;
  end proc;
```

EuclideanAlgorithm := **proc**(*a*::*integer*, *b*::*integer*)

local *c*, *d*, *r*;

c, *d* := abs(*a*), abs(*b*); **while** $d \neq 0$ **do** *r* := irem(*c*, *d*); *c*, *d* := *d*, *r* **end do**; *c*

end proc

```
> EuclideanAlgorithm(24,210);
```

$$6$$

Procedures may be nested, nested lexical scoping is used (a la Pascal).
Procedures may be returned and passed freely as parameters.
The simplest debugging tool is to insert print statements in the procedure. For example

```
> EuclideanAlgorithm := proc(a::integer,b::integer) local c,d,r;
      (c,d) := (abs(a),abs(b));
```

```
      while d <> 0 do r := irem(c,d); print(r); (c,d) := (d,r); od;
      c;
  end proc:
> EuclideanAlgorithm(24,210);
```

$$24$$
$$18$$
$$6$$
$$0$$
$$6$$

The next simplest debugging tool is the trace command. All assigment statements are displayed.

```
> trace(EuclideanAlgorithm);
```

$$\textit{EuclideanAlgorithm}$$

```
> EuclideanAlgorithm(24,210);
{--> enter EuclideanAlgorithm, args = 24, 210
```

$$c, d := 24, 210$$
$$r := 24$$
$$24$$
$$c, d := 210, 24$$
$$r := 18$$
$$18$$
$$c, d := 24, 18$$
$$r := 6$$
$$6$$
$$c, d := 18, 6$$
$$r := 0$$
$$0$$
$$c, d := 6, 0$$
$$6$$

```
<-- exit EuclideanAlgorithm (now at top level) = 6}
```

$$6$$

There is more. See **?proc** if you need more information or more tools.

```
>
```

Subscripted Names and String Utilities

Variables may be subscripted. For example, here is a polynomial in x1,x2,x3. You can assign to the subscripts.

```
> restart;
> f := 1-x[1]*x[2]*x[3];
> x[1] := 3;
```

$$f := 1 - x_1\, x_2\, x_3$$
$$x_1 := 3$$

```
> f;
```

$$1 - 3\, x_2\, x_3$$

There may be more than one subscript and the subscripts may be any value. For example, we may wish to record the numerical code for each letter of an alphabet. We could do this

```
> code["A"] := 0;
  code["B"] := 1;
```

$$code_{"A"} := 0$$

$$code_{"B"} := 1$$

etc. Basically, code is now a table (a hash table) with two entries. We can access an entry as follows

```
> code["B"];
```

$$1$$

```
> code["C"];
```

$$code_{"C"}$$

So given the following alphabet we can define the code of each letter in a loop. We also do the opposite.

```
> alphabet := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

$$alphabet := \text{"ABCDEFGHIJKLMNOPQRSTUVWXYZ"}$$

```
> N := length(alphabet);
```

$$N := 26$$

```
> for i from 0 to N-1 do
      x := alphabet[i+1];
      code[x] := i;
      char[i] := x;
   od:
```

We give procedures **encode** and **decode** to convert a string to a list of codes and back.

```
> encode := proc(s::string) local i;
     [seq( code[s[i]], i=1..length(s) )];
  end:
> decode := proc(x::list(integer)) local i,n;
     n := nops(x);
     cat( seq( char[x[i]], i=1..nops(x) ) );
  end:
> c := encode("BUYIBM");
```

$$c := [1, 20, 24, 8, 1, 12]$$

```
> decode(c);
```

$$\text{"BUYIBM"}$$

We can use tables to count the frequency of each letter in a piece of plaintext s as follows.

```
> s := "THISISTHEDAYTHATTHELORDHASMADE";
```

$$s := \text{"THISISTHEDAYTHATTHELORDHASMADE"}$$

First we initialize the table entries to 0. Then compute the frequencies

```
> F := table(): # create an empty table
  for i from 0 to 25 do F[char[i]] := 0 od:
> n := length(s);
  for i to n do x := s[i..i]; F[x] := F[x]+1; od:
```

$$n := 30$$

Now we convert to a list and use the sort command to sort in decreasing order. How to do this. We create a list of the form [[x1,f1], [x2,f2], ...] where each x is a letter and the f it's corresponding frequency then sort it then covert to probabilities.

```
> T := [seq( [char[i],F[char[i]]], i=0..25)];
```

$$T := [["A", 4], ["B", 0], ["C", 0], ["D", 3], ["E", 3], ["F", 0], ["G", 0], ["H", 5], ["I", 2], ["J", 0], ["K", 0], ["L", 1], ["M", 1], ["N", 0], ["O", 1], ["P", 0], ["Q", 0], ["R", 1], ["S", 3], ["T", 5], ["U", 0], ["V", 0], ["W", 0], ["X", 0], ["Y", 1], ["Z", 0]]$$

```
> sort(T, proc(x,y) if x[2]>y[2] then true else false fi end);
```

$$[["T", 5], ["H", 5], ["A", 4], ["S", 3], ["E", 3], ["D", 3], ["I", 2], ["Y", 1], ["R", 1], ["O", 1], ["M", 1], ["L", 1], ["Z", 0], ["X", 0], ["W", 0], ["V", 0], ["U", 0], ["Q", 0], ["P", 0], ["N", 0], ["K", 0], ["J", 0], ["G", 0], ["F", 0], ["C", 0], ["B", 0]]$$

```
> q := [seq( char[i-1]=evalf(T[i][2]/n,3), i=1..26 )];
```

$$q := [\text{"A"} = 0.133, \text{"B"} = 0., \text{"C"} = 0., \text{"D"} = 0.100, \text{"E"} = 0.100, \text{"F"} = 0., \text{"G"} = 0., \text{"H"} = 0.167,$$
$$\text{"I"} = 0.0667, \text{"J"} = 0., \text{"K"} = 0., \text{"L"} = 0.0333, \text{"M"} = 0.0333, \text{"N"} = 0., \text{"O"} = 0.0333, \text{"P"} = 0., \text{"Q"} = 0.,$$
$$\text{"R"} = 0.0333, \text{"S"} = 0.100, \text{"T"} = 0.167, \text{"U"} = 0., \text{"V"} = 0., \text{"W"} = 0., \text{"X"} = 0., \text{"Y"} = 0.0333, \text{"Z"} = 0.]$$

```
>
```

Polynomials and Finite Fields

Polynomials in Maple are simply input as formulae using the arithmetic operators. For example

```
> restart;
  x^4-3*x^2+12;
```

$$x^4 - 3x^2 + 12$$

is a polynomial in one variable, x with integer coefficients. Here is a polynomial in two variables.

```
> a := (x-y)*(x^2-y^2)*(x^3-y^3);
```

$$a := (x-y)(x^2-y^2)(x^3-y^3)$$

To multiply the factors of the polynomial out use the expand command

```
> expand(a);
```

$$x^6 - x^4y^2 + xy^5 - yx^5 + y^4x^2 - y^6$$

To factor the polynomial into prime factors with integer coefficients use the factor command

```
> factor(a);
```

$$-(y-x)^3(y+x)(y^2+yx+x^2)$$

We will only need polynomials in one variable and mostly work in the ring **Z**p[x] where p will be a prime integer. In what follows we show operations for **Z**p[x] and also **Q**[x]. For help for operations for polynomials see ?polynomial. For help for operations in **Z**p[x] see ?mod.

Here are two polynomials

```
> a := 2*x^6-3*x^5+3*x+3;
```

$$a := 2x^6 - 3x^5 + 3x + 3$$

```
> b := 3*x^4-4*x^3+1;
```

$$b := 3x^4 - 4x^3 + 1$$

The command **eval**(a(x), x=k) evaluates the polynomial a(x) at x = k. The command **Eval**(a, x=k) **mod** p does this modulo p . For example

```
> eval( a, x=2 );
```

$$41$$

```
> Eval( a, x=2 ) mod 7;
```

$$6$$

Here is how we can tabulate the values of this polynomial for all values in Z_7. We conclude that a(x) has no roots.

```
> seq( Eval(a,x=i) mod 7, i=0..6 );
```

$$3, 5, 6, 6, 4, 4, 5$$

We can interpolate a polynomial from it's values as follows

```
> a;
```

$$2x^6 - 3x^5 + 3x + 3$$

```
> X := [seq(i,i=0..8)];
```

$$X := [0, 1, 2, 3, 4, 5, 6, 7, 8]$$

```
> Y := [seq( eval(a,x=i), i=0..8 )];
```

$$Y := [3, 5, 41, 741, 5135, 21893, 70005, 184901, 426011]$$

```
> interp(X,Y,p);
```

$$2p^6 - 3p^5 + 3p + 3$$

The command **expand**(*a*b*) multiplies out the product *a b*. The command **Expand**(*a*b*) **mod** *p* does the product modulo *p*, that is, all coefficients in the resulting polynomial are reduced modulo *p* . For example

```
> p := 5;
```

$$p := 5$$

```
> expand(a*b);
```

$$6x^{10} - 17x^9 + 2x^6 + 12x^8 + 6x^5 - 3x^4 + 3x - 12x^3 + 3$$

```
> Expand(a*b) mod p;
```

$$x^{10} + 3x^9 + 2x^6 + 2x^8 + x^5 + 2x^4 + 3x + 3x^3 + 3$$

The operations **rem**(a,b,x) and **quo**(a,b,x) compute, respectively, the remainder *r* and quotient *q* of *a* divided by *b* satisfying $a = b\,q + r$ with $r = 0$ or $\deg(r) < \deg(b)$. The corresponding operations for Z_p are **Rem**(a,b,x) **mod** p and **Quo**(a,b,x) **mod** p . For example

```
> r := rem(a,b,x);
```

$$r := \frac{85}{27} + \frac{28}{9}x - \frac{2}{3}x^2 - \frac{16}{27}x^3$$

```
> q := quo(a,b,x);
```

$$q := \frac{2}{3}x^2 - \frac{1}{9}x - \frac{4}{27}$$

```
> expand( a = b*q+r );
```

$$2x^6 - 3x^5 + 3x + 3 = 2x^6 - 3x^5 + 3x + 3$$

```
> r := Rem(a,b,x) mod p;
```

$$r := 2x^3 + x^2 + 2x$$

```
> q := Quo(a,b,x) mod p;
```

$$q := 4x^2 + x + 3$$

```
> Expand( a = b*q+r ) mod p;
```

$$2x^6 + 2x^5 + 3x + 3 = 2x^6 + 2x^5 + 3x + 3$$

The commands **gcd**(a,b) and **lcm**(a,b) compute, respectively the greatest common divisor and least common multiple of two polynomials. The corresponding operations for Z_p are **Gcd**(a,b) **mod** p and **Lcm** (a,b) **mod** p. For example

```
> gcd(x^4-2*x^2+2,x^4+1);
```

$$1$$

```
> Gcd(x^4-2*x^2+2,x^4+1) mod p;
```

$$x^2 + 2$$

The command **gcdex**(a,b,x,'s','t') outputs $g = \mathrm{GCD}(a, b)$. It also outputs through the input parameters *s, t* integers satisfying the equation $s\,a + t\,b = g$ and satisfying $\deg(s) < \deg(b)$ and $\deg(t) < \deg(a)$. The corresponding command for Z_p is Gcdex(a,b,x,'s','t') mod p. For example

```
> gcdex(a,b,x,'s','t');
```

$$1$$

```
> s;
```

$$\frac{5024}{25565} - \frac{201}{25565}x + \frac{4958}{25565}x^3 - \frac{4734}{25565}x^2 + \frac{2208}{25565}x^5 - \frac{1188}{5113}x^4$$

```
> expand( s*a+t*b );
```

$$1$$

```
> Gcdex(a,b,x,'s','t') mod p;
```

$$x^2 + 3x + 1$$

```
> s;
```

$$x+4$$

```
> t;
```

$$x^3+3x^2+3x+4$$

```
> Expand(a*s+t*b) mod p;
```

$$x^2+3x+1$$

The command **irreduc**(a) outputs true if the polynomial a(x) is irreducible and the command **factor**(a) outputs the factorization of a(x) into irreducible factors over the integers. The corresponding commands for Z_p are **Irreduc**(a) **mod** p and **Factor**(a) **mod** p. For example

```
> factor(a);
```

$$2x^6-3x^5+3x+3$$

```
> factor(b);
```

$$(3x^2+2x+1)(x-1)^2$$

```
> Factor(a) mod 5;
```

$$2(x+1)(x+4)^5$$

```
> Factor(b) mod 5;
```

$$3(x+4)^2(x^2+4x+2)$$

The polynomial x^2+x+1 is irreducible modulo 2

```
> Factor(x^2+x+1) mod 2;
```

$$x^2+x+1$$

and hence the finite field of 4 elements can be represented by polynomials of degree < 2 over the integers modulo 2, i.e. the polynomials R = $\{0, 1, x+1, x\}$. We construct the multiplication table M for this finite field as follows.

```
> R := [0,1,x,x+1];
  M := matrix(4,4);
```

$$R := [0, 1, x, x+1]$$

$$M := \text{array}(1 .. 4, 1 .. 4, [\,])$$

```
> for i to 4 do
      for j to 4 do M[i,j] := Rem(R[i]*R[j],x^2+x+1,x) mod 2 od;
  od;
> print(M);
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & x & x+1 \\ 0 & x & x+1 & 1 \\ 0 & x+1 & 1 & x \end{bmatrix}$$

See **?mod** for other operations on polynomials over the integers modulo p .

```
>
```