http://www.mapleprimes.com/files/2816_rosenlib.zip

http://www.mhhe.com/math/advmath/rosen/r5/instructor/maple.html

Discrete Mathematics and Its Applications

Kenneth H. Rosen, AT&T Laboratories

The Maple Supplement

This book is a supplement to Ken Rosens's text Discrete Mathematics and its Applications, Fifth edition. It's entire focus is on the computational aspects of the subject. To make use of the code found in this supplement you need to make use of a special library that has been developed to supplement Maple for this book.

To make use of this code

1. Download the zip file containing the supplemental library by clicking this library link rosenlib.zip
2. Unzip the library in an appropriate location on your machine. This will create a subdirectory under the current directory with the name `rosenlib`
3. Add the rosenlib directory at the beginning of Maple's `libname` variable as in

   ```
   libname := "c:/rosenlib", libname:
   ```

   by placing this command in your `maple.ini` file, or somewhere near the top of your Maple worksheet.

4. Whenever you wish to use the code, load it first by executing the Maple command:

   ```
   with(Rosen);
   ```

   This will show you a list of the commands that are defined.

The sample code that is found throughout the text is found in blocks of code which begin with a reference to `libname`and then load this package.

A table of contents for the supplement appears below with hyperlinks directly to the Maple supplements relevant to the various chapters.

Discrete Mathematics and Its Applications

Kenneth H. Rosen, AT&T Laboratories

Chapter 1: The Foundations -- Logic and Proof, Sets, and Functions

Click here to access a summary of all the Maple code used in this section.

This chapter describes how to use *Maple* to study three topics from the foundations of discrete mathematics. These topics are logic, sets, and functions. In particular, we describe how *Maple* can be used in logic to carry out such tasks as building truth tables and checking logical arguments. We show to use *Maple* to work with sets, including how to carry out basic set operations and how to determine the number of elements of a set. We describe how to represent and work with functions in *Maple*. Our discussion of the topics in this chapter concludes with a discussion of the growth of functions.

1. Logic

Click here to access a summary of all the Maple code used in this section.

The values of true and false (T and F in Table 1 on page 3 of the main text) are represented in *Maple* by the words ***true*** and ***false***.

```
true, false;
```

Names can be used to represent propositions. If the truth value of **p** has not yet been determined, its value is just**p**. At any time, you can test this by entering the Maple expression consisting of only the name, as in

```
p;
```

A value can be assigned to a name by using the ***:=*** operator.

```
p := true;
```

Subsequently, every reference to **p** returns the new value of **p**, as in

```
p;
```

The value of **p** can be removed by assigning **p** its own name as a value. This is done by the statement

```
p := 'p';
```

The quotes are required to *stop***p** from evaluating.

The basic logical operations of *negation*, *Conjunction* (and), and *Disjunction* (or), are all supported. For example, we can write:

```
not p;
p and q;
p or q;
```

None of these expressions evaluated to ***true*** or ***false***. This is because evaluation can not happen until more information (the truth values of **p** and **q**) is provided. However, if we assign values to **p** and **q** and try again to evaluate these expressions we obtain truth values.

```
p := true:  q := false:
not p;
p and q;
p or q;
```

Maple does not support operations such as an *exclusive or* directly, but it can be easily programmed. For example, a simple procedure that can be used to calculate the *exclusive or* of two propositions is defined as:

```
XOR := proc(a,b)  ( a or b )  and not (a and b) end:
```

It is a simple matter to verify that this definition is correct. Simply try it on all possible combinations of arguments.

```
XOR( true , true );
XOR( true , false );
```

With the current values of **p** and **q**, we find that their exclusive or is true.

```
XOR( p , q );
```

1.1. Bit Operations

Click here to access a summary of all the Maple code used in this section.

We can choose to represent *true* by a 1 and *false* by a 0. This is often done in computing as it allows us to minimize the amount of computer memory required to represent such information.

Many computers use a 322 bit architecture. Each bit is a 0 or a 1. Each word contains 322 bits and typically represents a number.

Operators can be defined similar to **and** and **or** but which accept 1s and 0s instead of true and false. They are called bitwise operations. The bitwise *and* operator, **AND**, can be defined as

```
AND := proc( a , b )
  if a = 1 and a = b then 1
  else 0 fi;
end:
```

For example, the binary value of `AND(0,1)` is:

```
AND(0,1);
```

1.2. Bit Strings

Click here to access a summary of all the Maple code used in this section.

Once defined, such an operation can easily be applied to two lists by using the bitwise operation on the pair of elements in position 1, the pair of elements in position 2, and so on. The overall effect somewhat resembles the closing of a zipper and in Maple can be accomplished by using the command **zip**. For example, given the lists

```
L1 := [1,0,1,1,1,0,0]: L2 := [1,1,1,0,1,0,1]:
```

we can compute a new list representing the result of performing the bitwise operations on the pairs of entries using the command

```
zip( AND , L1 , L2 );
```

Beware! This direct method only works as intended if the two lists initially had the same length.
The **zip**command is used when you want to apply a function of two arguments to each pair formed from the members of two lists (or vectors) of the same length. In general, the call `zip(f, u, v)`, where **u** and **v** are lists, returns the list `f(u1, v1), f(u2, v2), ... , f(ulength(u), vlength(v))`. It allows you to extend binary operations to lists and vectors by applying the (arbitrary) binary operation coordinatewise.

1.3. A Maple Programming Example

Click here to access a summary of all the Maple code used in this section.

Using some of the other programming constructs in Maple we can rewrite **AND** to handle both bitwise and list based operations and also take into account the length of the lists.

We need to be able to compute the length of the lists using the `nops` command as in

```
nops(L1);
```

to take the maximum of two numbers, as in

```
max(2,3);
```

and to be able to form new lists. We form the elements of the new lists either by explicitly constructing the elements using the `seq` command or by using the `op` command to extract the elements of a list. The results are placed inside square brackets to form a new list.

```
L3 := [ seq( 0 , i=1..5) ]; L4 := [ op(L1) , op(L3) ];
```

We can use this to extend the length of short lists by adding extra 0s.

In addition, we use an *`if ... then`* statement to take different actions depending on the truth value of various tests. The **type** statement in Maple can test objects to see if they are of a certain type. Simple examples of such tests are:

```
type(3,numeric);
type(L3,list(numeric));
type([L3,L4] , [list,list] );
```

A new version of the **AND** procedure is shown below.

```
AND := proc(a,b)
  local i, n, newa, newb;
  if type([a,b],[list,list]) then
    n := max( nops(a),nops(b) );  # the longest list.
    newa := [op(a) , seq(0,i=1..n-nops(a)) ];
    newb := [op(b) , seq(0,i=1..n-nops(b)) ];
    RETURN( zip(AND,newa,newb) )
  fi;
  if type( [a,b] , [numeric,numeric] ) then
    if [a,b] = [1,1] then 1 else 0 fi
  else
    ERROR(`two lists or two numbers expected`,a,b);
  fi;
end:
```

Test our procedure on the lists **L1** and **L2**.

```
AND(L1,L2);
```

1.4. Loops and Truth Tables

Click here to access a summary of all the Maple code used in this section.

One of the simplest uses of *Maple* is to test the validity of a particular proposition. For example, we might name a particular expression as

```
e1 := p or q;
e2 := (not p) and (not q );
```

On input to *Maple* these simplify in such a way that it is obvious that `not e1` and **e2** will always have the same value no matter how **p** and **q** have been assigned truth values.

The implication p *implies* q is equivalent to *(not* p*) or* q, and it is easy to write a *Maple* procedure to compute the latter.

```
implies := (p,q) -> (not p) or q;
```

To verify that this Maple definition of `implies(p,q)` is correct examine its value for all possible values of p and q.

```
implies(false,false), implies(false,true);
implies(true,false),  implies(true,true);
```

A systematic way of tabulating such truth values is to use the programming loop construct. Since much of what is computed inside a loop is hidden, we make use of the **print** statement to force selected information to be displayed. We can print out the value of **p**, **q**, and `implies(p,q)` in one statement as

```
print(p,q,implies(p,q));
```

To execute this print statement for every possible pair of values for `p,q` by placing one loop inside another.

```
for p in [false,true] do
  for q in [false,true] do
    print( p , q , implies(p,q) );
  od:
od:
```

No matter how the **implies** truth values are computed, the *truth table* for the proposition *implies* must always have this structure.

This approach can be used to investigate many of the logical statements found in the supplementary exercises of this chapter. For example, the compound propositions such as found in Exercises 4 and 5 can be investigated as follows.

To verify that a proposition involving p and q is a tautology we need to verify that no matter what the truth value of p and the truth value of q, the proposition is always true. For example, To show that *((not*q*) and (*p *implies* q*)) implies (not* q*)* is a tautology we need to examine this proposition for all the possible truth value combinations of p and q. The proposition can be written as

```
p1 := implies( (not q) and implies(p,q) , not q );
```

For ***ptrue***, and ***qfalse***, the value of **p1** is

```
subs( p=true,q=false,p1);
```

The proposition **p1** is completely described by its truth table.

```
for p in [false,true] do
  for q in [false,true] do
    print(p,q,p1);
  od;
```

```
od;
```

When the variables **p** and **q** have been assigned values in the loop they retain that value until they are set to something else. Remember to remove such assignments by assigning **p** its own name as a value.

```
p := 'p'; q := 'q';
```

We can generate a *truth table* for binary functions in exactly the same manner as we have for truth tables. Recall the definition of **AND** given in the previous section. A table of all possible values is given by:

```
for i in [0,1] do
  for j in [0,1] do
    print(i,j,AND(i,j));
  od:
od:
```

We can even extend this definition of **AND** to one which handles pairs of numbers, or pairs of lists. The following procedure **AND2** accomplishes this.

```
AND2 := proc(a,b)
  if not type([a,b],
    [numeric,numeric],[list(numeric),list(numeric)])
    then RETURN('AND2'(a,b));
  fi;
  AND(a,b);
end:
```

Note that you can specify sets of types to **type**. As before, we have

```
AND2(0,0); AND2([0,1],[0,0]);
```

and when necessary, it can remain unevaluated as in

```
AND2(x,y);
```

Comparing Two Propositions

Truth tables can be also be used to identify when two propositions are really equivalent.

A second proposition might be

```
p2 := p1 and q;
```

To compare the truth tables for these two propositions (i.e. to test if they are equivalent) print out both values in a nested loop.

```
for p in [false,true] do
  for q in [false,true] do
    print(p,q,p1,p2);
  od;
od;
```

How would you test if **p2** was the same as `p implies q`?

1.5. Using *Maple* to Check Logical Arguments

Click here to access a summary of all the Maple code used in this section.

This section show you how to use some of *Maple*'s logical operators to analyze *real life* logical arguments. We'll need to make use of some of the facilities in the **logic** package. The **logic** package is discussed in detail in Chapter 9. To load the **logic** package, we use the **with** command.

```
with(logic):
```

In particular, we shall require the **bequal** function, which tests for the logical equivalence of two logical (boolean) expressions. Procedures in the **logic** package operate upon boolean expressions composed with the *inert* boolean operators **&and**, **&or**, **&not**, and so on, in place of ***and***, ***or***, ***not***. The inert operators are useful when you want to study the *form* of a boolean expression, rather than its value. Consult Chapter 9for a more detailed discussion of these operators.

A common illogicism made in everyday life, particularly favored by politicians, is confusing the implication$a$*implies* b with the similar implication *not* a *implies not* b. *Maple* has a special operator for representing the conditional operator '$\to$'; it is **&implies**. Thus, we can see the following in *Maple*.

```
bequal(a &implies b, &not a &or b);
```

Now, to see that '$a \to b$' and '$\bar{a} \to \bar{b}$' are *not* equivalent , and to further find particular values of $a$and b for which their putative equivalence fails, we can do the following.

```
bequal(a &implies b, (&not a) &implies (&not b), 'assgn');
assgn;
```

Another illogicism occurs when a conditional is confused with its converse. The ***converse*** of a conditional expression '$a \rightarrow b$' is the conditional expression '$b \rightarrow a$'. These are not logically equivalent.

```
bequal(a &implies b, b &implies a, 'assgn');
assgn;
```

However, a very useful logical principle is ***contraposition***, which asserts that the implication '$a \rightarrow b$' is equivalent to the conditional '$\overline{b} \rightarrow \overline{a}$'. You can read this as: a *implies* b is equivalent to *not* b *implies not* a, and you can prove it using *Maple* like this:

```
bequal(a &implies b, &not b &implies &not a);
```

For more discussion of the **logic** package, and of the so-called*inert* operators, see Chapter 9.

2. Quantifiers and Propositions

Click here to access a summary of all the Maple code used in this section.

Maple can be used to explore propositional functions and their quantification over a finite universe. To create a propositional function p such for which as $p(x) = x > 0$ in *Maple* we enter

```
p := (x) ->  x > 0;
```

The arrow notation -> is really just an abbreviated notation for constructing the *Maple* procedure `proc(x) x>0 end`. Once defined, we can use **p** to write propositions such as

```
p(x), p(3), p(-2) ;
```

To determine the truth value for specific values of **x**, we apply the **evalb** procedure to the result produced by **p**. as in **evalb(p(3))**.

We often wish to apply a function to every element of a list or a set. This is accomplished in *Maple* by using the**map** command. The meaning of the command **map(f,[1,2,3])** is best understood by trying it. To map **f** onto the list 1,2,3, use the command

```
map( f , [1,2,3] );
```

Each element of the list is treated, in turn, as an argument to f.

To compute the list of truth values for the list of propositions obtained earlier, just use **map**.

```
map( evalb, [ p(x),p(3),p(-2)] );
```

Note that the variable **x** has not yet been assigned a value, so the expression [redacted] does not yet simplify to a truth value.

Something similar can be done for multivariate propositional functions.

```
q := (x,y) -> x < y:
evalb( q(3,0) );
```

Maple can also be used to determine the truth value of quantified statements, provided that the universe of quantification is finite (or, at least, can be finitely parameterized). In other words, *Maple* can be used to determine the truth value of such assertions as *for all* x *in* S, [redacted], where S is a *finite* set. For example, to test the truth value of the assertion:*For each positive integer* x *less than or equal to* 100*, the inequality* $100x > 2^x$ *obtains.*where set is

```
S := seq(i, i = 1..10):
```

first generate the set of propositions to be tested as

```
p := (x) -> 100*x > 2^x:
Sp := map( p , S );
```

Next, compute the set of corresponding truth values.

```
Sb := map( evalb , Sp );
```

The quantified result is given by

```
if Sb = true then true else false fi;
```

A statement involving existential quantification, such as *there exists an* X *such that* [redacted], is handled in much the same way, except that the resulting set of truth values have less stringent conditions to satisfy. For example, to test the truth value of the assertion: *There is a positive integer* X *not exceeding* 100 *for which* $x^2 - 5$ *is divisible by* 111. over the same universe of discourse as before (the set **S** of positive integers less than or equal to100) construct the set of propositions and their truth values as before

```
q := (x) -> (irem(x^2 - 5, 11) = 0):
Sp := map(q,S);
Sb := map(evalb,Sp);
```

The **irem** procedure returns the integral remainder upon division of its first argument by its second. The existential test is just

```
if has( Sb , true ) then true else false fi;
```

To test different propositions, all you need do is change the universe **S** and the propositional function p.

If the universe of discourse is a set of ordered pairs, we can define the propositional function in terms of a list. For example, the function

```
q := (vals::list) -> vals[1] < vals[2]:
```

evaluates as

```
q( [1,30] );
```

A set of ordered pairs can be constructed using nested loops. To create the set of all ordered pairs [redacted] from the two sets, A and B use nested loops as in

```
A := 1,2,3: B := 30,60: S := NULL:
for a in A do
  for b in B do
    S := S , [a,b];
  od:
od:
```

The desired set is

```
S;
```

3. Sets

Click here to access a summary of all the Maple code used in this section.

As we have seen in the earlier sections, sets are fundamental to the description of almost all of the discrete objects that we study in this course. They are also fundamental to Maple. As such, Maple provides extensive support for both their representation and manipulation.

Maple uses curly braces (\{ , \}) to represent sets. The empty set is just

```
{};
```

A Maple set may contain any of the objects known to Maple. Typical examples are shown here.

```
1,2,3;
a,b,c;
```

One of the most useful commands for constructing sets or lists is the **seq** command. For example, to construct a set of squares modulo 57, you can first generate a sequence of the squares as in

```
s1 := seq( i^2 mod 30,i=1..60);
```

This can be turned into a set by typing

```
s2 := s1;
```

Note that there are no repeated elements in the set **s2**. An interesting example is:

```
seq(randpoly(x,degree=2),i=1..5);
```

The **randpoly** procedure creates a random polynomial of degree equal to that specified with the **degree** option (here 2). Thus, the last example above has generated a set consisting of 5 random quadratic polynomials in the indeterminate X.

The ordering of the elements is not always the same as the order you used when you defined the set. This is because *Maple* displays members of a set in the order that they are stored in memory (which is not predictable). By definition, the elements of a set do not appear in any particular order, and *Maple* takes full advantage of this to organize its storage of the sets and their elements in such a way that comparisons are easy for Maple. This can have some surprising consequences. In particular, you cannot sort a set. Use lists instead. If order is important, or if repeated elements are involved, use lists. Simple examples of the use of lists include

```
r := rand(100): # random no. < 100
L := [seq(r(), i=1..20)]; # list of 20 random nos. < 100
N := [1,1,1,2,2,2,3,3,3];
```

Such a lists can be sorted using the `sort` command.

```
M := sort(L);
```

The **sort** procedure sorts the list **L** producing the list **M** in increasing numerical order.

The number of elements in **N** is:

```
nops(N);
```

To find out how many distinct elements there are in a list simply convert it to a set, and compare the size of the set to the size of the original list by using the command `nops`.

```
NS := convert(N,set);
nops(NS);
```

Maple always simplifies sets by removing repeated elements and reordering the elements to match its internal order. This is done to make it easier for *Maple* to compute comparisons.

To *test* for equality of two sets write the set equation $A=B$ and can force a comparison using the `evalb` command.

```
A = B;
evalb( A = B );
```

3.1. Set Operations

Click here to access a summary of all the Maple code used in this section.

Given the two sets

```
A := 1,2,3,4; B := 2,1,3,2,2,5;
```

We can compute the relative difference of two sets using `minus`, as in

```
A minus B;
B minus A;
```

We can also construct their union

```
C := A union B;
```

Several other set operations are supported in *Maple*. For instance, you can determine the power set of a given finite set using the **powerset** command from the **combinat** package. To avoid having to use the**with** command to load the entire **combinat** package, you can use its *full name* as follows.

```
S := 1,2,3:
pow_set_S := combinat[powerset](S);
```

Try this with some larger sets.

The symmetric difference operator **symmdiff** is used to compute the symmetric difference of two or more sets. You will need to issue the command

```
readlib(symmdiff):
```

before you can use **symmdiff**. Then, the symmetric difference of **A** and **B** is

```
symmdiff(A, B);
```

Recall that the symmetric difference of two sets A and B is defined to be the set $(A \cup B) - (A \cap B)$ of objects that belong to exactly one of A and B.

```
symmdiff(A, B);
```

```
 (A union B) minus (A intersect B);
```

To construct the Cartesian product of two sets, we write a little procedure in *Maple* as follows. This procedure will construct the Cartesian product $A \times B$ of the two sets A and B given to it as arguments.

```
CartesianProduct := proc(A::set, B::set)
  local prod, # the Cartesian product; returned
         a,b; # loop variables
  prod := NULL; # initialize to a NULL sequence
```

loop like crazy

```
  for a in A do
    for b in B do
```

add the ordered pair [a,b] to the end

```
      prod := prod, [a,b];
    od;
  od;
  RETURN(prod); # return a set
end:
```

The procedure is called by providing it with two sets as arguments.

```
S := 1,2,3,4;
T := `Bill`, `Hillary`, `Chelsea`, `Socks`;
P := CartesianProduct(S, T);
```

Note that the order in which the arguments appear is relevant.

```
Q := CartesianProduct(T, S);
```

The representation and manipulation of infinite sets is somewhat more complicated. Discussion of this topic will occur in Chapter 10.

New sets and lists can also be created by mapping functions onto them. For example, you can map an unknown function onto a set, as in

```
s3 := map(f,s2);
```

Note that the ordering of the elements in **s3** need not have any relationship with the ordering of the elements in **s2**. Both are sets and order is irrelevant.

It may happen that **f** requires a second argument. If so, **map** can still be used as:

```
map(f, s2, y);
```

Again, the ordering is irrelevant, and in this case, because **f** is undefined, the result shows you explicitly what **map** has done.

You can also map onto lists. For example, given the list

```
l2 := convert(s2,list);
```

the list (in their correct order) of remainders of these numbers on division by 6 is just

```
map( modp , l2 , 6 );
```

where **modp** is a two argument procedure used to calculate remainder on division, as in

```
modp(23,6);
```

4. Functions and Maple

Click here to access a summary of all the Maple code used in this section.

For a discussion of the concept of mathematical functions see section 1.6 of the main text book. Functions are supported by Maple in a variety of ways. The two most direct constructs are tables and procedures.

4.1. Tables

Click here to access a summary of all the Maple code used in this section.

Tables can be used to define functions when the domain is finite and relatively small. To define a function using a table we must associate to each element of the domain an element of the codomain of this function.

A table **t** defining such a relationship can be defined by the command

```
t := table([a=a1,b=b1,c=c1]);
```

Once the table **t** is defined in this manner, the values of the expressions `ta`, `tb` and `tc` are

```
t[a];
```

```
t[b];
t[c];
```

The set of entries [redacted] occurring inside the square brackets form the domain of this discrete function. They are called *indices* in Maple. They can be found by using the **indices** command. For example, the *set* of indices of **t** is

```
idx := indices(t) ;
```

Each *index* is presented as a list. This is to allow for very complicated indices, perhaps involving pairs or triples such as `tx,y,z`.

In cases such as the above where the indices are simply names, the set of names can be recovered by applying a Maple procedure to every element of the set. Since

```
op( [a] );
```

evaluates to the single element contained inside this single element list `a`, we can recover the set of names by using the **map** command to apply the **op** command to every element of the set **idx**. This required command is:

```
map( op , idx );
```

The set [redacted] constitutes the *range* of the discrete function and can be recovered by the command **entries**, which returns the sequence of entries from a table, each represented as a list. To compute the range of a function represented by the table **t**, you can use **map** and **op** as before.

```
rng := map(op, entries(t));
```

The number of elements in the domain is just

```
nops(idx);
```

The number of elements in the range is just

```
nops(rng);
```

Adding New Elements

To add a new element to a table, simply use the assignment operator.

```
t[d] := d1;
```

Use the commands **indices** and **entries** to verify that this has extended the definition of **t**.

```
indices(t);
entries(t);
```

Tables versus Table Elements

You can refer to a table by either its name as in

```
t;
```

or its value as in

```
eval(t);
```

This crucial distinction is made because tables can have thousands of elements. It allows you to focus on the table as a single entity (represented by a name), or looking at all the detail through the elements themselves.

Defining Functions via Rules

Not all relations or functions are defined over finite sets. Often, in non-finite cases, the function is defined by a rule associating elements of the domain with elements of the range.

Maple is well suited for defining functions via rules. Simple rules (such as $x \longrightarrow x^2+3$) can be specified using Maple's [redacted] operator as in

```
(x) -> x^2 + 3;
```

Such a rules are very much like other Maple objects. They can be named, or re-used to form other expressions. For example, we name the above rule as f by the assignment statement

```
f := (x) -> x^2 + 3;
```

To use such a rule, we *apply* it to an element of the domain. The result is an element of the range. Examples of function application are:

```
f(3);
f(1101101);
```

We can even apply functions to indeterminates such as **t** as in

```
f(t);
```

The result is a formula which is dependent on **t**.

```
f(t);
```

You can even use an undefined symbol **g** as if it were a rule. Because the rule is not specified, the result returns unevaluated and in a form which can evaluate at some later time after you have defined a suitable rule. Examples of this include:

```
g(3);
g(t);
```

The ordered pair describing the effect of a function g on the domain element t is just:

```
[t,g(t)];
```

An algebra of functions

Just as for tables, functions can be manipulated by name or value. To see the current definition of a function use the `eval()` command, as in

```
eval(f);
```

If there is no rule associated with the name then the result will be a name.

```
eval(g);
```

Depending on how your Maple session is configured, you may need to issue the command

```
interface(verboseproc=2);
```

and then re-evaluate `eval(f)` before seeing the details of the function definition.

The **verboseproc**parameter controls how much information is displayed when an expression is evaluated. It is primarily used to view the source code for *Maple* library procedures, as shown above, but can be used to view the code for user functions, as well.

One advantage of being able to refer to functions by name only is that you can create new functions from old ones simply by manipulating them algebraically. For example the algebraic expressions

```
f + g;
g^2;
```

and

```
h := f^2;
```

each represent functions. To discover the rule corresponding to each of these new function definitions, simply apply them to an indeterminate. For these examples, we obtain

```
(f + g)(t);
(g^2)(t);
```

and

```
h(t);
```

Notice that in each case presented here, **g** is undefined so that `g(t)` is the algebraic expression that represents the result of applying the function **g** to the indeterminate **t**.

Even numerical quantities can represent functions. The rule

```
one := (x) -> 1;
```

simplifies to **1** and always evaluates to **1** when applied as a function.

```
one(t);
```

The result is that

```
(g +1)(t);
```

behaves exactly as if **1** were a function name for the function [redacted]. This generalizes to all numeric quantities. In particular `(3)*(x)` and `(3)(x)` behave very differently as the first one is multiplication and the second one is an application of the constant function [redacted]

```
(3)*x, (3)(x);
```

In both cases the parenthesis can be left off of the **3** with no change in the outcome.

4.2. Functional Composition

Click here to access a summary of all the Maple code used in this section.

Maple uses the **@** operator to denote functional composition. The composition $f \cdot g$ is entered in Maple as `f@g`. In a new Maple session the outcome of applying the function `h = f@g` to **t** is

```
restart;
h := f@g;
h(t);
```

Functions may be composed with themselves as in

```
g := f@@3;
```

The parenthesis around the exponent are important. They indicate that composition rather than multiplication is taking place. Again, this meaning becomes clear if you apply the function **g** to an unknown**t**, as in

```
g(t);
```

Constructing Functional Inverses

The identity function **Id** is the function

```
Id := (x) -> x;
```

A functional inverse of the function f is a function g that when composed with f results in the identity function [redacted]. For example, given

```
f := (x) -> 5*x^3 + 3;
```

the inverse of **f** is a function **g** such that

```
(f@g)(t) = t;
```

Use this equation to deduce that `g(t)` should be

```
isolate(%,g(t));
```

The right hand side of this equation can be used to actually define the function g. The Maple`unapply()` command can be used to turn the expression into a function. The righthand side is:

```
rhs(%);
```

To turn this expression into a rule, use `unapply()`, specifying the name used in the general rule as an extra argument.

```
g := unapply(%,t);
```

In this example, the resulting function is named **g**.

5. Growth of Functions

Click here to access a summary of all the Maple code used in this section.

The primary tool used to study growth will be plotting. This is handled in Maple by means of is shown here.

```
plot( ln(x),n, n*ln(n)  , n=2..6 );
```

For a single curve, omit the set braces, as in

```
plot( x^2 + 3 , x = 0..4 , y = 0..10);
```

The first argument to the plot command specifies the function or curve, or a set of curves. The second argument specifies a domain, while the optional third argument $y = a..b$ specifies a range. See the `plots` package for a wide variety of additional commands. Also, see the help page for *plot,options*.

It is possible to plot functions that are not continuous, provided that they are at least piecewise continuous. Two good examples relevant to discrete mathematics are the **floor** and **ceil** (*ceiling*) functions. Here, we plot both on the same set of axes.

```
plot(floor(x), ceil(x), x = -10..10);
```

6. Computations and Explorations

Click here to access a summary of all the Maple code used in this section.

1. What is the largest value of n for which $n!$ has fewer than 1000 decimal digits and fewer than 10000decimal digits?

Solution

The number of digits in a decimal integer can be determined in *Maple* by using the **length** function.

```
length(365);
```

To answer this question we can use the **length** function to construct a test for a ***while*** loop.

```
n := 1;
while length(n!) < 10 do
 n := n + 1;
od:
n - 1;
length((n - 1)!);
```

The same technique will allow you to find the largest n for which $n!$ has fewer than 1000 or fewer than 10000 digits.

2. Calculate the number of one to one functions from a set S to a set T, where S and T are finite sets of various sizes. Can you determine a formula for the number of such functions? (We will find such a formula in Chapter 4.)

Solution

We'll show here how to count the number of one to one functions from one finite set to another and leave the conjecturing to the reader. Since the number of one to one functions from one set to another depends only upon the sizes of the two sets, we may as well use sets of integers. A one to one function from a set S to a set T amounts to a choice of $|S|$ members of T and a permutation of those elements. Suppose that S has 3 members, we can view a one to one function from S to T as a labeling of 3 members of T with the members of S. That is we wish to choose 3 members of T and then permute them in all possible ways. We can compute these permutations with the function**permute** in the **combinat** package. If we assume that T has 5 members, then we enter the command

```
combinat[permute](5, 3);
```

The first arguments (here 5, the size of T) is the number of objects that we want to permute, and the (optional) second argument is the number of elements to permute. To count them, we use the**nops** routine.

```
nops(%);
```

If, instead, the set S had, say, 4 members, then we would compute:

```
nops(combinat[permute](5, 4));
```

3. We know that [redacted] is [redacted] when b and d are positive numbers with $d \geq 2$. Give values of the constants C and k such that $n^b \leq Cd^n$ whenever $n > k$ for each of the following sets of values: $b = 10$, $d = 2$; $b = 20$, $d = 3$; $b = 1000$, $d = 7$.

Solution

Here we solve only the last of these, leaving the rest for the reader. We are seeking values of the constants C and k such that $n^{1000} \leq C \cdot 7^n$ whenever $n > k$. We'll substitute a test value for$C$, and then use a loop to test for which values of n the inequality is satisfied.

```
n^1000 < C * 7^n;
```

```
left := lhs(%);
right := rhs(%%);
right_sub := subs(C = 2, right);
k := 1;
while evalb(subs(n = k,left) >= subs(n = k, right_sub)) do
  k := k +1 ;
od;
```

You should also try this for other test values of C.

You can also try to solve the equation $n^{1000} = C*7^n$ for n, using *Maple*s **solve** routine. For this particular example, you will need to use the help facility to learn more about the **W** function, which satisfies the equation $W(x)e^{W(x)} = x$. (It is a complex valued function, but is real valued for $x > 1/e$.)

7. Exercises/Projects

1. Use computation to discover what the largest value of n is for which $n!$ has fewer than 10000 digits.

2. Compare the rate of growth of factorials and the function f defined by $(x) \rightarrow e^{2x}$.

3. We saw that a list T with four elements had $24 = 4!$ permutations. Does this relationship hold true for smaller sets T? Go back and change the list T and re-compute the subsequent values. Does this relationship hold true for larger lists (say of size 5 or 6)? (Be careful as the number $n!$ grows very rapidly!)

4. Can you conjecture what the answer would be for larger n? Can you prove your conjecture? We will construct such a formula in Chapter 4.

5. Develop *Maple* procedures for working with fuzzy sets, including procedures for finding the complement of a fuzzy set, the union of fuzzy sets, and the intersection of fuzzy sets. (See Page 588 of the text.)

6. Develop *Maple* procedures for finding the truth value of expressions in fuzzy logic. (See Page 133 of the text.)

7. Develop maple procedures for working with multisets. In particular, develop procedures for finding the union, intersection, difference, and sum of two multisets. (See Page 577 of the text.)

Discrete Mathematics and Its Applications

Kenneth H. Rosen, AT&T Laboratories

Chapter 2. The Fundamentals -- Algorithms, the Integers and Matrices

Click here to access a summary of all the Maple code used in this section.

This chapter covers material related to algorithms, integers and matrices. An *algorithm* is a definite procedure that solves a problem in a finite number number of steps. In *Maple*, we implement algorithms using *procedures* that take input, process this input and output desired information.*Maple* offers a wide range of constructs for looping, condition testing, input and output that allows almost any possible algorithm to be implemented.

We shall see how to use *Maple* to study the complexity of algorithms. In particular, we shall describe several ways to study the time required to perform computations using *Maple*.

The study of integers, or number theory, can be pursued using *Maple*'s **numtheory** package. This package contains a wide range of functions that do computations in number theory, including functions for factoring, primality testing, modular arithmetic, solving congruences, and so on. We will study these and other aspects of number theory in this chapter.

Maple offers a complete range of operations to manipulate and operate on matrices, including all the capabilities discussed in the chapter of the text. In this chapter we will only touch on *Maple*'s capabilities for matrix computations. In particular we will examine matrix addition, multiplication, transposition and symmetry, as well as the meet, join and product operations for Boolean matrices.

1. Implementing Algorithms in *Maple*

Click here to access a summary of all the Maple code used in this section.

When creating algorithms, our goal is to determine a finite sequence of actions that will perform a specific action or event. We have already seen numerous examples of procedures or algorithms written in Maple{}. This chapter will provide further examples and touch on some of the built in procedures and functions that *Maple* provides that can make your task easier.

The following simple example serves to illustrate the general syntax of a procedure in Maple{}.

```
Alg1 := proc(x::numeric)
  global a; local b;
  a := a + 1;
  if x < 1 then b := 1;
  else b := 2; fi;
  a := a + b + 10;
  RETURN(a);
end:
```

A procedure definition begins with the key word ***proc*** and ends with the key word ***end***. The bracketed expression**(x::numeric)** immediately following ***proc*** indicates that one argument is expected whose value is to be used in place of the name X in any computations which take place during the execution of the procedure. The type**numeric** indicates that value that is provided during execution must be of type **numeric**. Type checking is optional.

The statement **global a;** indicates that the variable named **a** is to be borrowed from the main session in which the procedure is used. Any changes that are made to its value will remain in effect after the procedure has finished executing. The statement **local b;** indicates that one variable named **b** is to be *local* to the procedure. Any values that the variable **b** takes on during the execution of the procedure disappear after the procedure finishes. The rest of the procedure through to the ***end*** (the procedure body), is a sequence of statements or instructions that are to be carried out during the execution of the procedure. Just like any other object in *Maple*, a procedure can be assigned to a name and a colon can be used to suppress the display of the output of the assignment statement.

1.1. Procedure Execution

Click here to access a summary of all the Maple code used in this section.

Prior to using a procedure, you may have assigned values to the variables **a** and **b**, as in

```
a := x^2 ; b := 10;
```

The procedure is invoked (executed) by following its name with a parenthasized list of arguments. The argument cannot be anything other than a number. Otherwise an error will occur as in

```
Alg1(z);
```

For a numeric argument, the algorithm proceeds to execute the body of the procedure, as in.

```
Alg1(1.3);
```

Execution proceeds (essentially) as if you had replaced every occurrence of **x** in the body of the procedure by (in this case) 1.3 and then then executed the body statements one at a time, in order. Local variables have no value initially, while global variables have the value they had before starting execution unless they get assigned a new value during execution.

Execution finishes when you get to the last statement, or when you encounter a **RETURN** statement, which ever occurs first. The value returned by the procedure is either the last computed value or the value indicated in the **RETURN** statement. You may save the returned value by assigning it to a name, as in

```
 result := Alg1(1,3);
```

or you may use it in further computations, as in

```
Alg1(1,3) + 3;
```

It may happen that the procedure does not return any value. In fact, this can be done deliberately by executing the special statement `RETURN(NULL)`. This would be done if, for example, the procedure existed only to print a message.

1.2. Local and Global Variables

Click here to access a summary of all the Maple code used in this section.

What happens to the values of **a** and **b**?

Since **a** was global, any changes to its value that took place while the procedure was executing remain in effect. To see this, observe that the value of **a** has increased by 1.

```
a;
```

The variable **b** was declared local to the procedure body, so even though its value changed during execution, those changes have no effect on the value of the global variable **b**. To see this, observe that the global value of **b** remains unchanged at

```
b;
```

Local variables exist to assist with the actions that take place during execution of the procedure. They have no meaning outside of that computation.

During the execution of the body of the procedure, assignments are made, and the sequence of actions is decided by the use of two main control structures, loops and conditional branches.

A loop is a giant single statement which may have other statements inside it. Loops come in many forms. Two of the most common forms appear in the following sample procedures for printing numbers up to a given integer value.

The ***for*** loop typically appears as in

```
MyForLoop := proc(x::integer)
  local i;
  for i from 1 to x do
    print(i);
  od;
end:
```

The result of executing this procedure is

```
MyForLoop(3);
```

The ***while*** loop typically appears as in

```
MyWhileLoop := proc(x::integer)
  local j;
  j := 1;
  while j < x do
    print(j);
    j := j + 1;
  od;
end:
```

The result of executing this procedure is

```
MyWhileLoop(3);
```

In both cases, the statements that are repeated are those found between the two key words ***do*** and ***od***.

Conditional statements (based on the ***if*** statement) also come in several forms, the most common of which is illustrated in the procedure definition below. They form a single giant statement, each part of which may have its own sequence of statements. This functionality allows procedures to make decisions based on true or false conditions.

```
DecisionAlg := proc(y::integer)
  if (y< 5) then
    print(`The input is less than 5`);
  elif (y = 5) then
    print(`The number is equal to 5`);
  else
    print(`The number is larger than 5`);
fi;
end:
```

The outcome differs depending on what the argument value is at the time the procedure is invoked.

```
DecisionAlg(10);
DecisionAlg(5);
DecisionAlg(-1001);
```

This basic conditional statement forms the basis for decision making or *branching* in *Maple* procedures.

We now will combine aspects of all three of these programming tools to build a procedure directly from the problem statement stage, through the *pseudocode*, to the final *Maple* code.

Consider the following problem: *Given an array of integers, find and output the maximum and minimum elements.* So, an algorithm for solving this problem will take the form of inputing an array of integers, processing the array in some manner to extract the maximum and minimum elements, and then outputting these two values. Now, let's take what we have just outlined and make it more rigorous. That is, we wish to form *pseudocode*, which is not written in any specific computer language but allows easy translation into (almost) any computer language. In this case, we shall go point by point over the steps of our algorithm, called **MaxAndMin**. Again, the algorithm steps are constructed in a logical manner as follows:

1. The array is given as input. We call this input **t**.

2. We set the largest element and smallest element equal to the first element of the array.

3. We loop through the entire array, element by element. We call the current position **cur_pos**.

4. If the current element at **cur_pos** in the array is larger than our current maximum, we replace our current maximum with this new maximum.

5. If the element at **cur_pos** in the array is smaller than our current minimum, we replace our current minimum with this new minimum.

6. Once we reach the end of the array, we have compared all possible elements, so we output our current minimum and maximum values. They must be the largest and smallest elements for the entire array, since we have scanned the entire array.

Now, we convert each line of our pseudocode into *Maple* syntax. The reader should notice that each line of pseudocode translates almost directly into *Maple* syntax, with the keywords of the pseudocode line being the keywords of the *Maple* line of code.

To use the array functionality of *Maple*, we need to first load the **linalg** package. This loading outputs two warnings, which indicate that the two previous defined *Maple* functions for **norm** and **trace** have been overwritten with new definitions. Since these two functions will not be used in the following example, we can ignore the warnings and proceed with the procedure implementation.

```
with(linalg):
MaxAndMin := proc(t::array)
  local cur_max, cur_min, cur_pos;
  cur_max := t[1];
  cur_min := t[1];
  for cur_pos from 1 to vectdim(t) do
    if t[cur_pos] > cur_max then
      cur_max := t[cur_pos]
    fi;
    if t[cur_pos] < cur_min then
      cur_min := t[cur_pos]
    fi;
  od;
  RETURN([cur_min, cur_max]);
end:
```

We show the output of this procedure on two arrays of integers.

```
t := array(1..6, [1, 2, 45, 3, 2,10]);
r := array(1..5, [5, 4, 9, 10, 16]);
MaxAndMin(t);
MaxAndMin(r);
```

This example shows that the steps from pseudocode to *Maple* code are straightforward and relatively simple. However, keep in mind, many of these types of operations are already available as part of the*Maple* library. For example, the maximum of an array could be computed as in

```
tlist := convert( eval(t), list ):
max( op(tlist) );
```

2. Measuring the Time Complexity of Algorithms in *Maple*

Click here to access a summary of all the Maple code used in this section.

We are interested not only in the accuracy and correctness of the algorithms that we write, but also in their speed, or efficiency. Often, we are able to choose from among several algorithms that correctly solve a given problem. However, some algorithms for solving a problem may be more efficient than others. To choose an algorithm wisely requires that we analyze the efficiency of the various choices before us. This must be done in two ways: first, a mathematical analysis of the algorithm must be carried out, to determine its average and worst case running time; second, a practical implementation of the algorithm must be written, and tests made to confirm the theory. *Maple* cannot do the mathematical analysis for you, but it does provide several facilities for measuring the performance of your code. We shall discuss these facilities in this section.

First, note that *Maple* offers a way to measure the specific CPU (Central Processing Unit) time that a function used to compute a result. This is illustrated in the following example.

```
st := time():
MaxAndMin(r): MaxAndMin(t): MaxAndMin(t):
time()-st;
```

The **time** procedure reports the total number of seconds that have elapsed during the current *Maple* session. Here, we record the start time in the variable **st**, run the procedures that we wish to time, and then compute the time difference by calculating `time() - st`. This gives the time required to run the commands in seconds.

To illustrate this **time** function further, we will write a new **ManyFunctions** procedure that carries out some computations, but does not print any output. The reason is that our test case for output would normally output approximately 2 pages of digits, and this is not of interest to us here.

```
ManyFunctions := proc(x)
  local a,b,c,d,e;
  a := x;
  b := x^2;
  c := x^3;
  d := x!;
  e := x^x;
end:
st := time():
ManyFunctions(1000):
time() - st;
```

This standard technique for timing computations will be used occasionally throughout the remainder of the book.

Also, *Maple* allows use to keep track of any additions, multiplications and functions that we may wish to use, by way of the **cost** function. The following example illustrates its usage.

```
readlib(cost):
cost(a^4 + b + c + (d!)^4 + e^e);
```

We use the **readlib** command to load the definition of the**cost** function into the current *Maple* session. This is necessary for some of *Maple*'s library routines, but not for many. The help page for a particular function should tell you whether or not you need to load the definition for that function with a call to **readlib**.

So, the **cost** and **time** functions help measure the complexity a given procedure. Specifically, we can analyze the entire running time for a procedure by using the **time** command and we can analyze a specific line of code by using the **cost** command to examine the computation costs in terms of multiplications, additions and function calls required to execute that specific line of *Maple* code.

We will now use these functions to compare two algorithms that compute the value of a polynomial at a specific point. We would like to determine which algorithm is faster for different inputs to provide some guidance as to which is more practical. To begin this analysis, we construct procedures that implement the two algorithms, which are outlined in pseudocode on Page 1100 of the textbook.

```
Polynomial := proc(c::float, coeff::list)
  local power, i,y;
  power := 1;
  y := coeff[1];
  for i from 2 to nops(coeff) do
    power := power*c;
    y := y + coeff[i] * power;
  od;
  RETURN(y);
end:
```

```
Horner := proc(c::float, coeff::list)
  local power, i,y;
  y := coeff[nops(coeff)];
  for i from nops(coeff)-1 by -1 to 1 do
    y := y * c + coeff[i];
  od;
  RETURN(y);
end:
input_list := [4, 3, 2, 1];
Polynomial(5.0, input_list);
Horner(5.0, input_list);
```

In order to test these procedures, we need a sample list of coefficients. The following command generates a random polynomial of degree 1000 in x.

```
p2000 := randpoly(x,degree=2000,dense):
```

We have deliberately suppressed the output. Also, the algorithms expect a list of coefficients. This can be obtained from **p** as

```
q2000 := subs(x=1,convert(p2000,list)):
```

Now, using the *Maple* tools for measuring complexity, we determine which procedure runs relatively faster for a specific input.

```
st := time():
Horner(104567980000000.0, q2000 );
time() - st;
st := time():
Polynomial(104567980000000.0, q2000 );
time() - st;
```

Using *Maple*'s computational complexity analysis tools, we can determine that the implementation of Horner's method of polynomial evaluation is marginally quicker than the implementation of the more traditional method of substitution, for the input covered here.

3. Number Theory

Click here to access a summary of all the Maple code used in this section.

Maple offers an extensive library of functions and routines for exploring number theory. These facilities will help you to explore Sections 2.3, 2.4 and 2.5 of the text.

We begin our discussion of number theory by introducing modular arithmetic, greatest common divisors, and the extended Euclidean algorithm.

3.1. Basic Number Theory

Click here to access a summary of all the Maple code used in this section.

To begin this subsection, we will see how to find the value of an integer modulo some other positive integer.

```
5 mod 3;
10375378 mod 124903;
```

To solve equations involving modular congruences in one unknown, we can use the **msolve** function. For example, suppose we want to solve the problem: *What is the number that I need to multiply* 3 *by to get* 1*, modulo* 7*?* To solve this problem, we use the **msolve** function as follows.

```
msolve(3 * y = 1, 7);
```

So, we find that $3 \cdot 5 = 15 = 14 + 1 \equiv 1 \bmod 7$. Now, let us try to solve a similar problem, except that our modulus will be 6, instead of 7.

```
msolve(3 * y = 1, 6);
```

Now it appears that *Maple* has failed, but in fact, it has returned no solution, since no solution exists. In case there is any doubt, we will create a procedure to verify this finding.

```
CheckModSix := proc()
  local i;
  for i from 0 to 6 do
    print(i, 3 * i mod 6);
  od;
end:
CheckModSix();
```

We note that $3y \equiv 0 \bmod 6$ or $3y \equiv 3 \bmod 6$, and hence $3y \equiv 1 \bmod 6$ will never have a solution. This can be attributed to the fact that $\gcd(3,6)=1$. As one final example of solving congruences, we shall construct a problem that has multiple solutions.

```
msolve(4 * x = 4, 10);
```

3.2. Greatest Common Divisors and Least Common Multiples

Click here to access a summary of all the Maple code used in this section.

Maple provides a library routine **igcd** for computing the greatest common divisor of a set of integers. A few examples of the **igcd** function, along with other related functions, of *Maple* may be helpful.

```
igcd(3, 6);
igcd(6, 4, 12);
```

Here, we compute the greatest common divisor of the integers from 100 to 1000 inclusive.

```
igcd(seq(i, i = 10..100));
```

There is a related function **ilcm** that computes the least common multiple. The following examples illustrate its use.

```
ilcm(101, 13);
ilcm(6, 4, 12);
ilcm(seq(i, i = 10..100));
```

The last example calculates the least common multiple of the integers n in the range $10 \le n \le 100$.

Now to examine the relationships between least common multiples and greatest common divisors, we shall create a procedure called `IntegerRelations`.

```
IntegerRelations := proc(a,b)
  a*b, igcd(a,b), ilcm(a,b)
end:
IntegerRelations(6, 4);
IntegerRelations(18, 12);
```

These examples illustrate the relationship

$$ab = \gcd\{a,b\}\cdot\mathrm{lcm}\{a,b\}$$

for non-negative integers a and b

The **i** in **igcd** and **ilcm** stands for *integer*. The related functions **gcd** and **lcm** are more general and can be used to compute greatest common divisors and least common multiples of polynomials with rational coefficients. (They can also be used with integers, because an integer n can be identified with the polynomial $n\cdot x^0$.) The **igcd** and **ilcm** routines are optimized for use with integers, however, and may be faster for large calculations.

Now, having examined greatest common divisors, we may wish to address the problem of expressing a greatest common divisor of two integers as an integral combination of the integers. Specifically, given integers n and m, we may wish to express [redacted] as a linear combination of m and n, such as $x\cdot n + y\cdot m$, where x and y are integers. To solve this problem, we will use the Extended Euclidean algorithm from *Maple* contained in the function **igcdex**, which stands for Integer Greatest Common Divisor using the Extended Euclidean algorithm. Since the Extended Euclidean Algorithm is meant to return three values, the **igcdex** procedure allows you to pass two parameters as arguments into which the result will be placed. By quoting them, we ensure we pass in their names, rather than any previously assigned value. You can access their values after calling **igcdex**. This is illustrated in the following example.

```
igcdex(3,5, 'p', 'q');
p; q;
```

So, the desired linear combination is $2 \cdot 3 + (-1) \cdot 5 = 1 = \gcd(3,5)$. We continue with two more examples.

```
igcdex(2374, 268, 'x', 'y');
x; y;
igcdex(1345, 276235, 'a', 'b');
a; b;
```

3.3. Chinese Remainder Theorem

Click here to access a summary of all the Maple code used in this section.

Maple can be used to solve systems of simultaneous linear congruences using the Chinese Remainder Theorem. (See Page 1411 of the text.) To study problems involving the Chinese Remainder Theorem, and related problems, *Maple* offers the **chrem** function that computes the unique solution to the system

[redacted] ≡ [redacted]

[redacted] ≡ [redacted]

. — .

[redacted] ≡ [redacted]

of modular congruences. Specifically, we shall solve Example 5 (Page 1400 of the text) using the*Maple***chrem** function. Sun-Tzu's problem asks us to solve the following system of simultaneous linear congruences.

[redacted] ≡ [redacted]

[redacted] ≡ [redacted]

[redacted] ≡ [redacted]

The solution is easily computed in *Maple*, as follows.

```
chrem([2, 3, 2], [3, 5, 7]);
```

The first list of variables in the **chrem** function contains the integers $a_1, a_2, \ldots, a_n$ and the second list of variables contains the moduli $m_1, m_2, \ldots, m_n$. The following, additional example illustrates the use of non-positive integers.

[redacted] ≡ [redacted]

[redacted] ≡ $-8 \bmod 23$

[redacted] ≡ [redacted]

[redacted] ≡ [redacted]

```
chrem([34,-8,24,0],[98,23,47,39]);
```

Having covered gcd's, modularity, the extended Euclidean algorithm and the Chinese Remainder Theorem, we move to the problem of factoring integers, which has direct practical applications to *cryptography*, the study of secret writing.

3.4. Factoring integers

Click here to access a summary of all the Maple code used in this section.

To factor integers into their prime factors, the *Maple* number theory package **numtheory** must be loaded into memory, as follows.

```
with(numtheory):
```

If we wish to factor a number into its prime factors, we can use the *Maple***ifactor** command. For example, we can factor 1000 using **ifactor** as follows.

```
ifactor(100);
ifactor(12345);
ifactor(1028487324871232341353586);
```

By default, *Maple* uses the Morrison-Brillhart method, a factoring technique developed in the 1970's, to factor an integer into its prime factors. Beside using the Morrison-Brillhart method, *Maple* allows other methods of factorization to be used also. For instance, consider the following set of examples.

```
ifactor(1028487324871232, squfof);
ifactor(1028487324871232, pollard);
ifactor(1028487324871232, lenstra);
ifactor(1028487324871232, easy);
```

These examples illustrate several different methods of factorization available with *Maple*: the square-free method, Pollard's ρ method, and Lentra's elliptic curve method. The reader should explore these methods and various types of numbers that they factor efficiently or inefficiently. As an example, it is known that Pollard's method factors integers more efficiently if the factors are of the form $k \cdot m + 1$, where k is an optional third parameter to this method. It is left up to the reader to explore these alternative methods both using *Maple* and books on number theory.

The final factoring method which we will discuss, entitled **easy**, factors the given number into factors which are easy to compute. The following example illustrates this.

```
ifactor(1028487324871232341353586);
ifactor(1028487324871232341353586, easy);
```

The first method factors the given integer into complete prime factors, where as the second method factors the number into small components and returns `_c22` indicating the other factor has 222 digits and is too hard to factor. The time to factor is illustrated as follows.

```
st:=time():
ifactor(1028487324723234135358 6):
time()-st;
st:=time():
ifactor(1028487324723234135358 6, easy):
time()-st;
```

3.5. Primality Testing

Click here to access a summary of all the Maple code used in this section.

Finding large primes is an important task in RSA cryptography, as we shall see later. Here we shall introduce some of *Maple*'s facilities for finding primes. We have already seen how to factor integers using*Maple*. Although factoring an integer determines whether it is prime (since a positive integer is prime if it is its only positive factor other than 1), factoring is not an efficient primality test. Factoring integers with1000 digits is just barely practical today, using the best algorithms and networks of computers, while factoring integers with 2000 digits seems to be beyond our present capabilities, requiring millions or billions of years of computer time. (Here, we are talking about factoring integers not of special forms. Check out*Maple*'s capabilities. How large an integer can you factor in a minute? In an hour? In a day?)

So, instead of factoring a number to determine whether it is a prime, we use probabilistic primality tests.*Maple* has the **isprime** function which is based upon such a test. When we use **isprime**, we give up the certainty that a number is prime if it passes these tests; instead, we know that the probability this integer is prime is extremely high. Note that the probabilistic primality test used by **isprime** is described in depth in Kenneth Rosen's textbook *Elementary Number Theory and its Applications* (3rd edition, published by Addison Wesley Publishing Company, Reading, Massachusetts, 1992).

We illustrate the use of the **isprime** function with the following examples.

```
isprime(101);
isprime(2342138342111);
isprime(23218093249834217);
```

Since this number is not too large for us to factor, we can use **ifactor** to check the result.

```
ifactor(23218093249834217);
```

The *Maple* procedure **ithprime** computes the ith prime number, beginning with the prime number 2.

```
ithprime(1);  # the first prime number
ithprime(2);  # the second prime number
ithprime(30000);
```

The function **ithprime** produces prime numbers that are guaranteed to be prime. For small prime numbers, it simply looks up the result in an internal table, while for larger arguments, it operates recursively. This function should be used when an application needs to be certain of the primality of an integer and when speed is not an over-riding consideration.

In addition to these two procedures, *Maple* provides the **nextprime** and **prevprime** functions. As their names suggest, they may be used to locate prime numbers that follow or precede a given positive integer. For example, to find the first prime number larger than 10000, we can type

```
nextprime(1000);
```

Similarly, the prime number before that one is

```
prevprime(%);
```

Note that each of **nextprime** and **prevprime** is based on the function **isprime**, so their results are also determined probabilistically.

In general, to see the algorithm used by a procedure, set the **interface** parameter **verboseproc** equal to2 and calling **eval** on the procedure. For example,

```
interface(verboseproc=2);
eval(nextprime);
```

These procedures provide several ways to generate sequences of prime numbers. A guaranteed sequence of primes can be generated quite simply using **seq**.

```
seq(ithprime(i), i=1..100);  # the first 100 primes
```

3.6. The Euler ϕ -Function

Click here to access a summary of all the Maple code used in this section.

The Euler ϕ function $\phi(n)$ counts the number of positive integers not exceeding n that are relatively prime go n. Note that since $\phi(n) = n - 1$ if, and only if, n is prime, we can determine whether$n$ is prime by finding $\phi(n)$. However, this is not an efficient test.

In *Maple*, we can use the function **phi** in the **numtheory** package in the following manner.

```
phi(5);
phi(10);
phi(107);
```

This tells us that there are 4 numbers less than 5 that are relatively prime to 5, implying that 5 is a prime number. Since $\phi(5) = 4$ and $\phi(107) = 106$, we see that 5 and 1077 are primes.

If we wished to determine all numbers $k_1, k_2, \ldots, k_m$ that have $\phi(k_i) = n$, we can use the **invphi**function of *Maple*. For example, to find all positive integers k such that $\phi(k) = 2$, we need only compute

```
invphi(2);
```

4. Applications of Number Theory

Click here to access a summary of all the Maple code used in this section.

This section explores some applications of modular arithmetic and congruences. We discuss hashing, linear congruential random number generators, and classical cryptography.

4.1. Hash Functions

Click here to access a summary of all the Maple code used in this section.

Among the most important applications of modular arithmetic is *hashing*. For an extensive treatment of hashing, the reader is invited to consult Volume 3 of D. Knuth's *The Art of Computer Programming*.

Hashing is often used to improve the performance of search algorithms. This is important in many software systems such as in databases, and in computer languages translators (assemblers, compilers, and so on).*Maple* itself relies extensively, in its internal algorithms, upon hashing to optimize its performance.

Often, in a software system, it is necessary to maintain a so-called *symbol table*. This is a table of fixed size in which various objects, or pointers to them, are stored. The number of objects input to the system is, in principle, unlimited, so it is necessary to map objects to locations in the symbol table in a many-to-one fashion. For this, a hashing function is used. Many types of hashing functions are used, but among the most effective are those based on modular arithmetic. Here, we'll look at how a hash function of the kind discussed in your textbook might be used in a simple minded way in the management of a simple symbol table. Our symbol table routines will do nothing more than install and search a table by means of a hash function.

The first thing to do is to decide on the size of the symbol table. We'll use *Maple*'s **macro** facility to introduce a symbolic constant for this.

```
macro(hashsize = 101); # a prime number
```

Thus, our symbol table will have a fixed number **hashsize** of entries, not all of which need be occupied at a given time.

For this simple example, a *symbol* will simply be a string consisting exclusively of uppercase letters. For the symbol table itself we shall use a *Maple* array.

```
symtab := array(1..hashsize);
```

We'll define the hash function **Hash** to be used shortly, but first let's take a look at two procedures that will call **Hash**. The first is the function **Install**, used to enter a string into the symbol table.

```
Install := proc(s::string)
  local hashval;
  global symtab;
  hashval := Hash(s);
  symtab[hashval] := s;
end:
```

This procedure returns nothing; it is called only for the side effect of inserting the string argument into the symbol table. The second function is **Lookup**, used to search the symbol table for a string.

```
Lookup := proc(s::string)
  local hashval, i;
  hashval := Hash(s);
  if symtab[hashval] = s then
    RETURN(symtab[hashval]);
  else
    RETURN(NULL);
  fi;
end:
```

The function **Lookup** computes the hash value of its argument, and returns the data stored at that address in the symbol table.

Now let's take a look at a hash function for strings that is based on modular arithmetic. We'll use a variant of the simple hash function discussed in the text. A very effective hash function for integers may be obtained by computing the remainder upon division by some modulus. Here, we'll use this idea by assigning to a string consisting of uppercase letters of the alphabet an integer, and then computing its value modulo the size of the symbol table. For this reason, we have chosen a symbol table size that is a prime number to maximize the *scattering* effect of the hash function, thus reducing the likelihood of collisions. (A major defect of our routines is the lack of any collision resolution strategy. You are asked in the exercises to repair this deficiency.) To compute an integer encoding of a string, we shall need the following procedure**UpperToAscii** that assigns to an uppercase character its ASCII value. First, we define a function**UpperToNum** that assigns to each uppercase character a number based on its position in the alphabet. This is not necessary here, but we'll reuse this function later on in this section.

```
alias( I = I );
alias( E = E );
alphabet := [A, B, C, D, E, F, G, H, I, J, K, L, M,
             N, O, P, Q, R, S, T, U, V, W, X, Y, Z]:
for i from 1 to nops(alphabet) do
  UpperToNum(op(i, alphabet)) := i - 1:
od:
```

Notice the special treatment given here to **I** and **E**. Each is a special symbol to *Maple*; **E** is used to denote the base of the natural logarithm, while **I** represents the imaginary unit. The **alias** calls above remove these special meanings. (In general, you should be very careful about how you redefine symbols having special meaning to *Maple*. We can do this here because we are certain that we do not need these two symbols to have their special meaning.) Here, now, is the ASCII conversion routine.

```
UpperToAscii := proc(s::string)
  if not length(s) = 1 then
    ERROR(`argument must be a single character`);
  fi;
```

The ASCII value of 'A' is 65.

```
  RETURN(65 + UpperToNum(s));
end:
```

Here, finally, is the hash function.

```
Hash := proc(s::string)
  local hashval, # return value
        i;       # loop index
  hashval := 0;
```

Sum the ASCII values of the characters in s

```
  for i from 1 to length(s) do
    hashval := hashval + UpperToAscii(substring(s, i..i));
  od;
```

Compute the residue

```
  hashval := hashval mod hashsize;
  RETURN(hashval);
end:
```

We can see some of the hash values computed by our hashing function as follows.

```
Hash(MATH);
Hash(ALGEBRA);
Hash(FUNWITHMAPLE);
```

Now, a program might use the symbol table routines that we have developed here as follows. Given a list of strings to process in some way, the strings can be entered into the symbol table in a loop of some kind.

```
Input := [`BILL`, `HILLARY`, `CHELSEA`, `SOCKS`, `BILL`];
for s in Input do
  if Lookup(s) = NULL then
    Install(s);
  fi;
od;
```

Here is what the symbol table looks like now that some entries have been installed.

```
eval(symtab);
```

Each question mark (?) represents a cell in the table that is not yet occupied; its location is displayed as a subscript. The contents of occupied cells are shown here as strings.

To later extract strings from the symbol table for processing, or to determine whether a given string is already present in the table, the function **Lookup** is used.

```
Lookup(`BILL`);
Lookup(`GEORGE`);
```

4.2. Linear Congruential Pseudorandom Number Generators

Click here to access a summary of all the Maple code used in this section.

Many applications require sequences of random numbers. They are important in cryptology and in generating data for computer simulations of various kinds. Often, random number streams are used as input to routines that generate random structures of different kinds, such as graphs or strings. It is impossible to produce a truly random stream of numbers using software only. (Software employs algorithms, and anything that can be generated by an algorithm is, by definition, not random.) Fortunately, for most applications, it is sufficient to generate a stream of **pseudorandom** numbers. This is a stream of numbers that, while not truly random, does nevertheless exhibit some of the same properties of a truly random number stream. Effective algorithms for generating pseudorandom numbers can be based on modular arithmetic. We examine here an implementation of a linear congruential pseudorandom number generator. This generates a sequence [redacted] of numbers [redacted] satisfying the system of equations

$$x_{n+1} = ax_n + c \bmod m$$

where a, c and the modulus m are some integer constants. Here, the first term [redacted] of the sequence is initialized to some convenient value called the *seed*. One advantage the a pseudorandom number generator has in certain applications is that it can be reproduced simply by using the same seed. This is useful when the results are being used for test data that needs to be replicable from one instance of the test to the next.

To implement the generator, we start with a subroutine that does the modular arithmetic for us.

```
NextVal := proc(x,a,c,m)
  RETURN((a * x + c) mod m);
end:
```

This simply computes the next value in the sequence, given the current value in the argument **x**. The generator itself is fairly simple. It simply initializes the data associated with the system, and constructs a loop to append the requested number of terms to the sequence returned. Our procedure **LCPRNG** takes two arguments, the length of the list to generate, and a *seed* or starting value used to initialize the generator.

```
LCPRNG := proc(n::integer, seed::integer)
  local prn_list, # list of pseudorandom numbers to return
        modulus,
        multiplier,
        increment,
        i,x;        # temporaries
```

these could be globals instead, or passed as parameters

```
  multiplier := 7^5;
  modulus := 2^31 - 1;
  increment := 66;
  prn_list := NULL;
  x := seed;
  for i from 1 to n do
    prn_list := prn_list, x;
    x := NextVal(x, multiplier, increment, modulus);
  od;
  prn_list := [prn_list];
  RETURN(prn_list);
end:
```

To generate a list of 5 pseudorandom numbers, with seed 3, you can simply type:

```
LCPRNG(5, 3);
```

In practical use, you would likely choose the seed in a somewhat random fashion, say, based on the time of day. For instance, the following little routine produces an integer based on the CPU time of your *Maple*session.

```
SeedIt := proc()
  trunc(1000 * time());
end;
```

You can use it to generate somewhat unpredictable seed values for **LCPRNG**, as follows.

```
LCPRNG(5, SeedIt());
```

4.3. Classical Cryptography

Click here to access a summary of all the Maple code used in this section.

We are going to examine here a way to implement an affine cipher in *Maple*. We'll need to convert between letters and numbers, as the ciphers we'll examine are based on arithmetic modulo 266. The **UpperToNum**function presented in the discussion of hashing in the previous section will serve well in one direction, but we shall also require its inverse **NumToUpper**.

```
for i from 0 to nops(alphabet) - 1 do
  NumToUpper(i) := op(i + 1, alphabet):
od:
```

A general affine cipher has the form

$$f(p) = (ap + b) \bmod 26$$

where the pair [redacted] is the *key* to the cipher. The argument p is the integer code for some plain text that is to be encrypted. For decryption to be feasible, the key must be chosen so that f is a bijection. This amounts to choosing a to be relatively prime to the modulus 266.

We shall use a helper function **CryptChar** to process a single character.

```
CryptChar := proc(s::string, key::[integer, integer])
  local mult, # the multiplier
        trans;# the translator
  if not length(s) = 1 then
    ERROR(`argument must be a single character`);
  fi;
  mult := key[1];
  trans := key[2];
  RETURN(NumToUpper((UpperToNum(s) * mult + trans) mod 26));
end:
```

This procedure encrypts single characters.

The cipher itself simply loops over all the character in the string input, and passes the individual calculations to **Cryptchar**.

```
AffineCypher := proc(s::string, key::[integer, integer])
  local i,         # loop variable
        multiplier,
        translator,
        ciphertext;# the encrypted text
  ciphertext := NULL;
  for i from 1 to length(s) do
    ciphertext := cat(ciphertext,
         CryptChar(substring(s, i..i), key));
  od;
  RETURN(ciphertext);
end:
```

Let's see how this works with a very regular string **ABCDE** and a few different keys:

```
AffineCypher(ABCDE, [1,3]); # Caesar cipher
AffineCypher(ABCDE, [3,0]);
AffineCypher(ABCDE, [3,3]);
```

Try this with various other keys. To encrypt the *Maple* string **MATHISFUN**, using the key [redacted], we can type

```
AffineCypher(MATHISFUN, [3,2]);
```

An important observation is the the decryption function for an affine cipher is itself another affine cipher.

Suppose that we are encrypting data with the key [redacted].

```
AffineCypher(ABC, [7,2]);
```

To decipher the cipher text, we need to compute the inverse cipher. This is just the affine cipher with key [redacted], as later computations will show.

```
AffineCypher(CJQ, [15,22]);
```

Computing the key to the inverse cipher involves solving an affine congruence, modulo the alphabet size (here, 266). For our example, the encryption key was [redacted], corresponding to the congruence

$$y \equiv 7x + 2 \bmod 26$$

To compute the key for the decryption function, we need to solve this equation for x, modulo 266. We can use the **solve** procedure to do this in *Maple*, as follows.

```
x := 'x': y := 'y':
e := y = 7 * x + 2;
solve(e, x);
```

```
f := x = % mod 26;
```

Thus, the defining equation for the inverse cipher is

$$y \equiv 15x + 22 \bmod 26$$

to which corresponds the decryption key [redacted].

5. RSA Cryptography

Click here to access a summary of all the Maple code used in this section.

We shall now show how to use *Maple* to implement the RSA cryptosystem. We shall use *Maple* to construct keys, and to encrypt and decrypt messages.

To construct keys in the RSA system, we need to find a pair of large primes, say, with 1000 digits each. We shall explain how to do this later in this section. Since messages can be decrypted by anyone who can factor the product of these primes, the two primes must be large enough so that their product is extremely difficult to factor. A 2000 digit integer fits the bill since factoring requires an extremely large amount of computer time.

Because the use of very large prime numbers would make our examples impractical *as examples*, we shall illustrate the RSA system using smaller primes, and then discuss, separately, how you can use *Maple* to generate large prime numbers.

Implementing the RSA system involves two steps.

1. Key generation
2. The encryption algorithm

Let us first consider key generation. The first step is to choose two distinct, large prime numbers, p and q, each of about 1000 digits. From these, we must produce the public key, which consists of the public modulus $n = pq$, and the public exponent e, as well as the private key, consisting of the public modulus n, and the inverse of $e$modulo $\phi(n)$, where ϕ is Euler's ϕ-function. (The public modulus n is not really a part of the private key, but it does no harm to include it, and makes the implementation of the encryption engine below a little cleaner.) Since e is unrelated to the primes p and q, it can be generated in a number of ways. Two popular choices for e in real systems are 3 and the 4th Fermat number $F_4 = 2^{2^4} + 1 = 65537$. Another approach is to generate a random prime number that does not divide $\phi(n)$. For our simple implementation below, we shall simply take e to be the constant 133. Here is a *Maple* procedure to handle key generation for us.

```
GenerateKeys := proc(p::integer, q::integer)
  local n,    # public modulus
        e,    # public exponent
        d,    # d * e = 1 (mod phin)
        phin; # phi(n) = (p - 1)(q - 1)
  n := p * q; # Compute the public modulus
  phin := (p - 1) * (q - 1);
  e := 13; # This could be generated randomly
```

Compute d such that e * d = 1 (mod phin)

```
  d := op(1, op(Roots(e * x - 1) mod phin));
  RETURN([[n, e], [n, d]]);
end:
```

This function returns both the public and private keys in a two member list of the form `public_key, private_key`. Note that each entry is itself a two member list of integers. In fact, it is useful to introduce a *Maple* type for keys, both public and private. This allows us to write clean code and still have procedures check their arguments.

```
`type/rsakey` := proc(obj)
   type(obj, [posint, posint]);
end:
```

Thus, to generate keys using the prime numbers $p = 43$ and $q = 59$, we can type

```
keys := GenerateKeys(43, 59);
```

and retrieve the public key and private key pairs, using **op**.

```
public_key := op(1, keys);
private_key := op(2, keys);
```

Our type definition above allows us to test the type of an object to determine whether it has the form of an RSA public or private key.

```
type(public_key, rsakey);
type(private_key, rsakey);
```

In a practical RSA implementation, we would likely use some of the techniques discussed at the end of this section to incorporate into our **GenerateKeys** procedure the generation of the primes p and q as well, rather than passing them as arguments.

Now that we have seen how to generate RSA keys, we shall discuss an implementation of the encryption engine of the RSA scheme. The code is fairly simple.

```
RSA := proc(key::rsakey, msg::list(posint))
  local ct, # cipher text; returned
        pe, # public exponent
        pm, # public modulus
        i;  # loop index
```

Extract key information

```
  pm := key[1];
  pe := key[2];
  ct := [];
  for i from 1 to nops(msg) do
    ct := [op(ct), msg[i]^pe mod pm];
  od;
  RETURN(ct);
end:
```

The first argument to **RSA** is the key. The second argument is the message to be processed, in the form of a list of positive integers.

Let's look at how we can use **RSA** to transmit a message securely. First, a message in English or any other natural language must be encoded as a list of positive integers. To encode an English message, we can assign to each letter of the alphabet a two digit number, indicating the position of the letter in the alphabet. (So, for instance, $A \rightarrow 01$, $B \rightarrow 02$, and so on.) Then we break the resulting string of digits into blocks of four digits each. The list consisting of these four digit blocks will be the input to **RSA**. The block length must be chosen so that, after conversion, the largest integer produced is less than the modulus n. Here, we have $n = 2537$, and the largest block that can be produced is 25255 for *ZZ*.

For a specific example, consider the message *STOP HERE*. This is encoded as the list [redacted].

Now, to transmit the message *STOP HERE*, the sender uses his or her private key **private_key**, computed above, to encrypt the message as follows.

```
ciphertext := RSA(private_key, [1819, 1415, 0805, 1705]);
```

The recipient receives the encrypted text **ciphertext**. To decrypt it, the sender's public key is used, as follows.

```
RSA(public_key, ciphertext);
```

The resulting list is decoded as the original message *STOP HERE*.

5.1. Generating Large Primes

Click here to access a summary of all the Maple code used in this section.

If we were to use small prime numbers p and q such as those used in our earlier example, there would be no real security. Anyone could factor n, the product of these primes, and then easily find the decrypting key d from the encrypting key e. However, using *Maple*'s prime generation and factoring facilities, we can generate fairly large prime numbers for use in an RSA public key. Remember that what is needed is a pair of prime numbers, each of about 1000 digits. Moreover, they should be selected in an unpredictable fashion. To do this in *Maple*, we can use the **rand** procedure to produce a random 1000 digit number, and then use the **nextprime** function to choose the smallest prime number that exceeds it. This will guarantee that the prime number has at least 1000 digits.

For example, to choose two prime numbers of 1000 digits each, we can proceed as follows.

```
BigInt := rand(10^99..10^100): # get a random 100 digit number
a := nextprime(BigInt());
b := nextprime(BigInt());
```

Note that even generating such large prime numbers is not cheap.

```
st := time():
nextprime(BigInt()):
time() - st;
```

It is left to the reader to incorporate these ideas in an improved version of our **GenerateKeys** procedure.

6. Base Expansions

Click here to access a summary of all the Maple code used in this section.

To convert numbers from a representation in one base to a representation in another base, the *Maple* procedure**convert** can be used. For example, to convert the number 222 to its base 2 (binary) representation, we can type

```
convert(22, base, 2);
```

while the hexadecimal representation can be obtained by using the command

```
convert(22, base, 16);
```

Conversion to a nonstandard base, such as 733 is also possible:

```
convert(1237765, base, 73);
```

These examples illustrate that the output of the conversion is a list of elements ordered from lowest base value (the "ones" column) on the left, followed by the increasingly higher powers on the right. Converting integers to one of the *standard* bases from base 100 is fairly simple. The following examples illustrate some of these conversions.

```
convert(22, binary);
convert(34, octal);
convert(1050, hex);
```

Used in this form, the **convert** command provides more readable output, but it is limited to the commonly used bases shown here. Also, we can convert from a non-decimal base into base 100, using the *standard* bases of binary, octal, and hexadecimal.

```
convert(1001001,  decimal, binary);
convert(`42E`, decimal, hex);
```

To convert from arbitrary bases into a decimal value, we shall construct a function that takes the list output that`convert(int, base, n)` returns.

```
MyConvert:=proc(L::list, base::integer)
  local i, dec_value;
  dec_value:=0;
  for i from 1 to nops(L) do
    dec_value := dec_value + L[i]*(base^(i-1));
  od;
end:
t:=convert(145743, base, 73);
MyConvert(t, 73);
```

The examples we have given illustrate that we may convert from any integer base into decimal, and from decimal to any integer base.

Maple can do arbitrary base conversions, at the cost of simplicity of representation (for bases larger than 266, we run out of *digits*). For arbitrary base conversions, both input and output integers are represented as (*Maple*) lists of digits. For example, the base 100 integer 19966 is represented as the list 1, 9, 9, 6. Each such digit must be a non-negative integer less than the input base, and each digit is interpreted as a base 100 integer. For instance, the base 100 number 2635183311 can be written as a polynomial in 266 as

$$11 + 2 \cdot 26^2 + 17 \cdot 26^3 + 4 \cdot 26^4 + 22 \cdot 26^5$$

and is represented as the list 11, 0, 2, 17, 4, 22 of its *digits* in base 266. To convert this number to, say, base999, we can type the following.

```
convert([11, 0, 2, 17, 4, 22], base, 26, 99);
```

The output should be interpreted as a list of digits for the base 999 number

$$32 + 87 \cdot 99^1 + 57 \cdot 99^2 + 73 \cdot 99^3 + 2 \cdot 99^4$$

The same thing can be done with any pair of bases.

7. Matrices

Click here to access a summary of all the Maple code used in this section.

Maple provides several matrix construction mechanisms and numerous operations to manipulate them. The simplest way to construct a matrix in *Maple* is by representing each row as a list and using the **matrix** command from the**linalg** package.

```
with(linalg):
a1:=matrix(
  [[1, 2, 3, 4, 5], [6, 5, 4, 3, 2],
   [2, 4, 6, 8, 0], [9, 7, 5, 3, 1]]
);
```

Matrix expressions can be constructed as in

```
2*a1 + 3;
```

To complete the operations and view the results use the **evalm** function as in

```
evalm(2*a1+3);
```

Maple also allows us to create matrices of common types, such as symmetric, antisymmetric, diagonal, identity or sparse.

```
a2:=array(1..5, 1..5, identity);
evalm(a2);
a3:=array(1..4, 1..4,
  [(1,1)=4, (2, 2)=90, (3,3)=-34, (4,4)=103],
  diagonal);
```

We may also exponentiate a square matrix any number of times, as shown by the following two examples.

```
evalm(a3^2);
evalm(
  matrix([[1,5,3], [-6, -4, 3], [0, 107, 4]])^5
);
```

As a note of caution, since matrix multiplication is non-commutative, *Maple* has a special multiplication function designed specifically for matrices, whose name is *. The usage is outlined in the following example.

```
m1:=matrix([[1,86,3],[52, -3, 8], [-5, 34, 12]]);
m2:=matrix([[4, 0, 6], [ 2, 7, -4], [18, 5, 13]]);
evalm(m1 &* m2);
evalm(m2 &* m1);
```

We now will examine how *Maple* can be used to solve a problem similar to that outlined in Example 6 on page 1544of the text. To solve this question, we first create a procedure that will keep track of the number of steps required to multiply two matrices.

```
MyMatrixMult:=
  proc(A::matrix, B::matrix, prev_steps::integer)
  local i, j, k, q, C, steps;
  steps:=prev_steps;
  C:=matrix(rowdim(A), coldim(B), zeroes);
  for i from 1 to rowdim(A) do
  for j from 1 to coldim(B) do
    C[i,j]:=0;
    for q from 1 to coldim(A) do;
      C[i,j]:=C[i,j]+A[i,q]*B[q,j];
      steps:=steps+1;
    od;
  od;
  od;
  [evalm(C), steps];
end:
MyMatrixMult(
  matrix(4, 3, [1,0,4,2,1,1,3,1,0,0,2,2]),
  matrix(3, 2, [2,4,1,1,3,0]),
  0 );
```

Having created this algorithm, we now will examine the best possible arrangement of matrix products so that computation is kept to a minimum. To make our sample size of matrices fairly large, we will use the **randmatrix**command, which produces matrices filled with random integers.

```
A1:=randmatrix(20,15):
A2:=randmatrix(15,35):
A3:=randmatrix(35,10):
MyMatrixMult(
  A1,
  MyMatrixMult(A2, A3, 0)[1],
  MyMatrixMult(A2, A3, 0)[2])[2];
MyMatrixMult(
  MyMatrixMult(A1, A2, 0)[1],
  A3,
  MyMatrixMult(A1, A2, 0)[2])[2];
```

7.1. Zero-One Matrices

Click here to access a summary of all the Maple code used in this section.

Using *Maple*, we can create and manipulate zero-one matrices in a manner similar to integer valued matrices. In our exploration of zero-one matrices, we will create a zero-one matrix in a form that can be manipulated in *Maple*, then proceed to create the meet, join and Boolean product functions for zero-one matrices.

To create a Boolean matrix, we will define **1** to be ***true***, and **0** to be ***false***. This will allow *Maple* to apply boolean functions on each element in the matrix via the **bsimp** function of the logic package, as illustrated in the next example.

```
with(logic):
bsimp(true &and false);
bsimp(true &or false);
```

We now move on to constructing a boolean matrix. To do this, we will use the **matrix** function as was used before, entering the matrix as in zero-one form, then converting it to a *Maple* boolean form, using the**map** command.

```
with(linalg):
B1:=matrix(3,3,[1,0,1,1,1,0,0,1,0]);
int_to_bool(1):=true:
int_to_bool(0):=false:
bool_to_int(true):=1:
bool_to_int(false):=0:
B2:=map(int_to_bool, B1);
map(bool_to_int, B2);
```

Having created a boolean matrix, both in zero-one format and *Maple*`true/false` format, we shall now create procedures for the boolean meet and join, as outlined on Page 1577 of the textbook.

```
BoolMeet:=proc(A::matrix, B::matrix)
  local i, j, C;
  C:=matrix(rowdim(A), coldim(A), zeroes);
  for i from 1 to rowdim(A) do
    for j from 1 to coldim(A) do
    C[i,j]:=bsimp(int_to_bool(A[i,j]) &and int_to_bool(B[i,j]));
    od:
  od;
  map(bool_to_int,C);
end:
B3:=matrix(3, 3, [1, 0,0,1,1,1,0,0,0]);
BoolMeet(B1, B3);
BoolJoin:=proc(A::matrix, B::matrix)
  local i, j, C;
  C:=matrix(rowdim(A), coldim(A), zeroes);
  for i from 1 to rowdim(A) do
    for j from 1 to coldim(A) do
    C[i,j]:=bsimp(int_to_bool(A[i,j]) &or int_to_bool(B[i,j]));
    od;
  od;
  map(bool_to_int,C);
end:
BoolJoin(B1, B3);
```

Having implemented the Boolean join and meet function, we conclude this subsection on zero-one matrices by implementing the Boolean product function, which is outlined on Page 1588 of the text.

```
BoolProd:=proc(A::matrix, B::matrix)
  local i, j, k, C;
  C:=matrix(rowdim(A), coldim(B), zeroes);
  for i from 1 to rowdim(A) do
    for j from 1 to coldim(B) do
      C[i,j]:=false;
      for k from 1 to coldim(A) do
        C[i,j]:=bsimp(
          C[i,j]
          &or (int_to_bool(A[i,k])
               &and int_to_bool(B[k,j])
              )
                     );
      od;
    od;
  od;
  map(bool_to_int, C);
end:
I1:=matrix(3, 2, [1,0,0,1,1,0]);
I2:=matrix(2, 3, [1,1,0,0,1,1]);
I3:=BoolProd(I1, I2);
```

8. Computations and Explorations

Click here to access a summary of all the Maple code used in this section.

For this section on the *Computations and Explorations* section of the textbook, we shall cover five representative questions; those being questions 1, 3, 4, 6, and 7, dealing with material on factorization and primality testing.

1. Determine whether $2^p - 1$ is prime for each of the primes p not exceeding 1000.

Solution

To solve this problem, we'll write a *Maple* program that tests each prime less than or equal to a given to see whether $2^p - 1$ is prime, and produces a list of those primes for which it is. This is a good example of the use of the **nextprime** routine for looping over the list of primes up to some value.

```
Q1 := proc(n::integer)
   local cur_prime, mlist;
   cur_prime := 2;
   mlist := NULL;
   while cur_prime <= n do
     if isprime(2^cur_prime - 1) then
       mlist := mlist, cur_prime;
     fi;
     cur_prime := nextprime(cur_prime);
   od;
   mlist := [mlist];
   RETURN(mlist);
end:
```

To check the primes not exceeding 1000, we type the following.

```
Q1(100);
```

For another approach, consider the **mersenne** procedure from the **numtheory** package; it is based on a table lookup algorithm. Using it, you can compute the 100th Mersenne prime, for example, with a call like.

```
numtheory[mersenne]([10]);
```

Because it relies on a table in the *Maple* library, you cannot access very large Mersenne primes. See the help page for **numtheory,mersenne** for more information on this routine.

It is of note that there is a better test, called the *Lucas-Lehmer test*, that is more efficient and can be implemented in *Maple*. For a complete description of the algorithm, consult Rosen's text on Number Theory (as referenced earlier in the chapter).

2. Find as many primes of the form n^2+1 where n is a positive integer as you can. It is not know whether there are infinitely many such primes.

Solution

Looking at this question, we again construct a *Maple* procedure to aid in our search of primes of this form.

```
Q3 := proc(n)
  local i;
  for i from 1 to n do
  if isprime(i^2 + 1) then
    printf(`For i = %d, the number i^2 + 1 = %d is prime`,
           i, i^2 + 1);
  fi;
  od;
end:
```

To save space, we'll only do a small calculation.

```
Q3(10);
```

You can try `Q3(1000)` or `Q3(10000)`, which yield several pages of output. What can you observe about the output?

3. Find 100 different primes, each with 1000 digits.

Solution

Maple's **nextprime** function makes this too easy, so we shall construct 100*random* primes of 1000digits each. It is still not too hard, though; the only extra wrinkle is that we must guard against including a prime in our list more than once. Our procedure here is a little more general, since it is almost trivial to have it take, as an argument, the number of primes to produce.

```
PrettyBigPrimes := proc(howmany::integer)
  local ptab,  # table of 10 primes
        BigInt,# random 10 digit number generator
        n,     # loop variable
        p;     # index into ptab
  BigInt := rand(10^99..10^100);
  ptab := {};
  for n from 1 to howmany do
    p := nextprime(BigInt());
```

loop until we get a new one

```
    while member(p, ptab) do
       p := nextprime(p);
    od;
    ptab := ptab union p;
  od;
```

convert to a list and return it

```
  RETURN(convert(ptab, `list`));
end:
```

To save space, we'll show the output of calculating only 3 primes.

```
PrettyBigPrimes(3);
```

However, to illustrate that this is a fairly expensive computation, we show a timed run for a calculation of 100 primes.

```
st := time():
PrettyBigPrimes(10):
time() - st;
```

4. Find a prime factor of each of 100 different 200-digit odd integers, selected at random. Keep track of how long it takes to find a factor of each of these integers. Do the same thing for 100 different 300-digit odd integers, 100 different 400-digit odd integers, and so on, continuing as long as possible.

Solution

For this question, we first need to use *Maple* to select random numbers in a specific size range; that is, random 200-digit numbers (and 300-digit and 400-digit and so on). To do this, we will use the**rand** function of *Maple*.

```
Twenty:=rand(10^19..10^20):
Twenty();
```

The next step in solving Question 6 is to create some pseudocode that may help us in determining the desired results:

1. Create a random number
2. Verify the number is odd. If the random number is even, simply add 1 to it to make it odd.
3. Determine a factor of the random number (we need not find all prime factors).
4. Record the time required, and return to step (a)

We have already outlined how to do step (a), and step (b) can be fulfilled by the use of the **mod**function. We can accomplish step (c) by the use of the **ifactor** function, and step (d) can be done using the *Maple***time** function. We now move on to defining the procedure that is required.

```
RandFactor := proc(num_digits::integer)
  local i, temp, temp_fact,
  total_time, Generator, st;
  total_time := 0;
  Generator := rand(10^(num_digits-1)..10^num_digits);
  for i from 1 to 10 do
    temp := Generator();
    if (temp mod 2 = 0) then
      temp := temp+1;
    fi;
    st := time();
    temp_fact := ifactor(temp, easy);
    total_time := total_time + time() - st;
    print( temp = temp_fact );
  od;
  printf(`: %a`, total_time);
end:
RandFactor(20);
```

The reader can continue this exploration with 300-digit numbers and upwards, to determine the size of input that makes *Maple* factoring impractical.

5. Find all pseudoprimes to the base 2, that is, composite integers n such that $2^{n-1} \equiv 1 \bmod n$, where $n$does not exceed 100000?

Solution

Using *Maple*, this problem is relatively straightforward to solve. We can use a ***for***-loop structure to increment our n variable, the ***isprime*** function to determine which values of n are composite, and the ***mod*** function to calculate the remainder of 2^{n-1} modulo n.

The *Maple* code is shown below. Because of the length of the output, we show here only the first1000 values. You can check values up to 10000, or even higher, yourself.

9. Exercises/Projects

Click here to access a summary of all the Maple code used in this section.

1. Test which is faster for computing the greatest common divisor of a collection of integers, the **igcd** or **gcd**function.

2. How would you use *Maple* to generate the list of the first 100 prime numbers larger than one million?

3. Investigate further the comparative performance of Horner's method (discussed in Section 2.22) and the method of substitution for polynomial evaluation.

4. Use *Maple* to find the one's complement of an arbitrary integer. (See Page 1355 of the text.)

5. For which odd prime moduli is -1 a square? That is, for which prime numbers p does there exist an integer $x$for which $x^2 \equiv -1 \bmod p$?

6. Use *Maple* to determine which numbers are perfect squares modulo n for various values of the modulus n. What conjectures can you make about the number of different square roots an integer has module n? (Hint: Use the *Maple* function **msqr**).

7. Use *Maple* to find the base 2 expansion of the 4th Fermat number $F_4 = 2^{2^4} + 1$. Do the following for several large integers n. Compute the time required to calculate the remainder modulo n, of various bases $b$raised to the power [redacted] (that is, to calculate [redacted]) using two different methods: First, do the calculation by a straightforward exponentiation; then do it using the binary expansion of [redacted] with repeated squarings and multiplications. Why do you think it is a good choice for the public exponent in the RSA encryption scheme?

8. Modify the procedure **GenerateKeys** that we developed to produce the keys for the RSA system to incorporate the techniques for generating large random prime numbers. Make your procedure take as an argument a *security* parameter, as measured by the number of digits in the public modulus.

9. Write *Maple* routines to encode and decode English sentences into lists of integers, as described in the section on RSA encryption. (You may ignore spaces and punctuation, and assume that all letters are uppercase.)

10. Modify the symbol table management routines presented in this chapter to employ a collision strategy. Instead of storing the string data itself in the symbol table, use a list, and search the list linearly for a given input, after computing the location in the table by hashing. A reference for information on collision resolution strategies is Section 4.44 in K.

Rosen, *Elementary Number Theory and its Applications,* 3*rd ed.*, Addison-Wesley, Reading, Massachusetts, 1992.

11. (Class Project) The *Data Encryption Standard* (DES) specifies a widely used algorithm for private key cryptography. (You can find a description of this algorithm in Cryptography, Theory and Practice, by Douglas Stinson, published by CRC Press). Implement the DES in *Maple*.

12. There are infinitely many primes of the form $4n+1$ and infinitely many of the form $4N+3$. Use *Maple* to determine for various values of X whether there are more primes of the form $4n+1$ less than X than there are primes of the form $4n+3$, or vice versa. What conjectures can you make from this evidence?

13. Develop a procedure for determining whether Mersenne numbers are prime using the Lucas-Lehmer test as described in number theory books such as K. Rosen, *Elementary Number Theory and its Applications,* 3*rd ed.*, Addison-Wesley, Reading, Massachusetts, 1992. How many Mersenne numbers can you test for primality using *Maple*?

14. ***Repunits*** are integers with decimal expansions consisting entirely of 1s (e.g. $111, 1111111$, and 11111111111). Use *Maple* to factor repunits. How many prime repunits can you find? Explore the same question for repunits in different base expansions.

15. Compute the sequence of pseudorandom numbers generated by the linear congruential generator $x_{n+1} = (ax_n + c) \bmod m$ for various values of the multiplier a, the increment c, and the modulus m. For which values do you get a period of length m for the sequence you generate? Can you formulate a conjecture about this?

16. The *Maple* function **tau** (in the **numtheory** package) implements the function τ defined, for positive integers n by declaring that $\tau(n)$ is the number of positive divisors of n.

```
numtheory[tau](20);
```

Use *Maple* to study the function τ. What conjectures can you make about τ? For example, when is $\tau(n)$ odd? Is there a formula for $\tau(n)$? For which integers m does the equation $\tau(n) = m$ have a solution, for some integer n? Is there a formula for $\tau(mn)$ in terms of $\tau(m)$ and $\tau(n)$?

17. Develop a procedure that solves Josephus' problem. This problem asks for the permutation describing the order in which soldiers are killed when n soldiers arrange themselves around a circle and repeatedly execute every mth soldier, given the integers m and n.

Discrete Mathematics and Its Applications

Kenneth H. Rosen, AT&T Laboratories

Chapter 3. Mathematical Reasoning, Induction and Recursion

Click here to access a summary of all the Maple code used in this section.

In this chapter we describe how *Maple* can be used to help in the construction and understanding of mathematical proofs. Computational capabilities may seem not particularly relevant to the study of proofs. However, in reality, these capabilities can be helpful with proofs in many ways. In this chapter we describe how *Maple* can used to work with formal rules of inference. We describe how*Maple* can be used to help gain insight into constructive and non-constructive proofs. Moreover, we show how to use*Maple* to help develop proofs using mathematical induction, even demonstrating how *Maple* can be used both for the basis step and the inductive step in the proof of summation formulae. Moreover, we show how *Maple* can be used to compute terms of recursively defined sequences. We will also compare the efficiency of generating terms of such a sequence via inductive versus recursive techniques.

1. Methods of Proof

Click here to access a summary of all the Maple code used in this section.

Although *Maple* cannot take theorems and output proofs for those theorems, it can take logical expressions and simplify them or determine characteristics such as whether a boolean expression is satisfiable or is a tautology. To work with logical expressions in *Maple*, we shall need to use some of the facilities provided by the **logic** package, a topic discussed more fully in Chapter 9.

Firstly, we will examine the logical operators of*and*, *or*, *not* and *implies*. There is no system operator ***implies***; to study conditionals, we must work with the inert boolean operators provided by the **logic** package. These all begin with the character `\mexpr{&}` so, for instance, we use **&and** instead of ***and***, and **&not** in place of ***not***. See Chapter 9 for a more complete discussion of the distinction between these two sets of operators. Here are some examples of the use of the inert boolean operators.

```
with(logic):
a := &not(x1 &and x2);
b := (x3 &or x4) &implies (x5 &and x6);
```

We concern ourselves now with determining how *Maple* simplifies boolean expressions if we have them in combination. We begin with a simple *double negative* example.

```
c := &not (&not x7);
```

This can be simplified by the use of the **bsimp** function of*Maple*.

```
bsimp(c);
```

Now, we move to a more complex example, in which the reader can confirm the correctness of the simplification, constructing a truth table.

```
d := &not (&not x8 &and &not x9);
bsimp(d);
```

The next example illustrates the simplification of Modus Ponens. We first state that p *implies* q and p *is true*.

```
e := (p &implies q) &and (p);
```

Then we simplify the boolean expression,

```
bsimp(e);
```

determining that q *and* p *is true*, and since we already knew p *was true*, we have concluded that q *is true*.

The **bsimp** function is a general simplifier for boolean expressions constructed using the inert boolean operators. It computes a simplified boolean expression equivalent to its argument. Consult Chapter 9 for more details on **bsimp**.

We can also use *Maple* to determine if an expression is a tautology, by using the **tautology** function in the "logic" package.

```
tautology(x1 &and x2);
tautology((&not x3) &or x3);
```

We now show how *Maple* can be used to gain more insight into some constructive proofs. Specifically, we will examine how *Maple* can be used to explore the conclusions of Example 200 (page 1799) in the text, which is exploring how to construct a list of sequential composite numbers. We shall create the constructive algorithm outlined in the text, in order to explicitly generate this list of composite number.

```
MakeComposite := proc(n::integer)
  local x,i, L;
  L := {};
  x := (n + 1)! + 1:
  for i from 1 to n do
     L := L union (x + i);
  od;
  L;
end:
MakeComposite(5);
MakeComposite(11);
```

While *Maple* can be used to generate the list of n consecutive composite integers generated by the proof, it is not possible to use *Maple* to derive the proof itself. It should be noted that this argument does not provide the smallest set of n consecutive composite integers. However, given a positive integer n, you could use *Maple* to find the smallest sequence of n consecutive composite integers. (See Problem 3 in the Computations and Explorations section of the text, and the exercises at the end of this chapter.)

Now, we turn our attention to Example 211, which is the non-constructive existence proof of the fact that there are an infinite number of prime numbers. Now, since this proof is non-constructive, we cannot simply create an algorithm to generate a *larger prime* assuming the existence of a *largest prime*.

However, the key idea in the proof is to consider the primality of the integer $n! + 1$, and *Maple* can be used to pursue this a little further. Of course, it is possible that $n! + 1$ is itself a prime number but, even if it is not, its smallest prime factor must be larger than n. We can use *Maple* to find this smallest prime factor by factoring $n! + 1$ directly, using the *Maple*library routine **ifactor**. Let's examine a few of the numbers of this form.

```
for n from 1 to 10 do
  ifactor(n! + 1);
od;
```

We can see from the output that, while some of these numbers are themselves prime, others are not, and from this, we can read off the smallest of their prime factors.

To determine the least prime factor of each of these integers, we can write a routine as follows.

```
with(numtheory): # define 'factorset'
LeastFactor := proc(n::integer)
  min(op(factorset(n)));
end:
```

This uses the procedure **factorset**, from the **numtheory** package, to compute the set of factors of the integer input, and then simply selects its least member.

```
for n from 1 to 10 do
 LeastFactor(n! + 1);
od;
```

Now, we confront our final example of using *Maple* for exploring mathematical theorems. In this case, we will examine Goldbach's conjecture: that is, *Every even integer greater than 4 can be expressed as a sum of two primes.*

```
Goldbach := proc(p::integer)
  local i,j,finished, next_i_value;
  finished := false;
  i := 0; j := 0;
  while not finished do
    next_i_value := false;
    i := i+1;  j := i;
    while not next_i_value do
      if ithprime(i) + ithprime(j) = p then
        printf(`%d can be expressed as %d + %d`,
               p, ithprime(i), ithprime(j));
        finished := true;
        next_i_value := true;
      fi;
      j := j+1;
      if ithprime(j) >= p then
        next_i_value := true
      fi;
    od;
  od;
end:
Goldbach(12);
Goldbach(24);
```

Now, we create a procedure to examine the Goldbach conjecture more automatically.

```
ManyGoldbach := proc(startval::integer,finalval::integer)
  local i;
  for i from max(2, startval) to finalval do
    Goldbach(2 * i);
  od;
end:
ManyGoldbach(2,4);
ManyGoldbach(20, 26);
```

2. Mathematical Induction

Click here to access a summary of all the Maple code used in this section.

It is possible to use *Maple* to assist in working out proofs of various mathematical assertions using mathematical induction. In fact, with *Maple* as your assistant, you can carry out the entire process of discovery and verification interactively. We'll demonstrate this here, first with a very simple example, to highlight the steps involved; then we'll examine a somewhat less trivial problem.

It is likely that one among the first examples of the use of mathematical induction that you encountered is the verification of the formula

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

for the sum of the first n positive integers. *Maple* is ideally suited to proving formulae, such as this one, because the steps in an inductive proof involve symbolic manipulation. It is hardly necessary in this simple example, but you can use *Maple* to generate a large body of numerical data to examine.

```
seq(sum(i, i = 1..n), n = 1..30);
```

By generating a sufficiently large set of numerical data, and with a little insight, you should eventually be able to guess the formula above. The output shows that [redacted] is a little less than twice the sum of the first n integers for the values of n tested. From the pattern we see, we might guess that the correct formula is a quadratic function of n, solve for the coefficients, and then test whether this procedure produces the correct formula.

A useful technique for experimenting with such guesses is to generate lists of pairs consisting of the sequence you are interested in and various *guesses* that you come up with. To investigate the hypothesis that the formula is quadratic, you might start by generating a list of pairs such as the one that follows.

```
s := 's':
s := proc(n::integer)
 local i;
 sum(i, i = 1..n);
end:
seq([s(n), n^2], n=1..20);
```

To explore whether the sum is a quadratic function of n, we can enter a generic quadratic in n, and solve for the coefficients a, b and c.

```
n := 'n'; # remove any value
q(n) := a * n^2 + b * n + c;
```

We need three equations to solve for three coefficients.

```
eqns := seq(subs(n = k, q(n)) = s(k), k = 1..3);
```

Now we instruct *Maple* to solve these equations for the three coefficients.

```
solve(eqns, a, b, c);
```

Our original formula then becomes

```
subs(%, q(n));
```

At this point, you can use *Maple*s ability to manipulate expressions *symbolically* to help to construct an inductive proof. Here is how an interactive proof of the formula above, by mathematical induction, can be carried out in*Maple*.

The general term of the sum is

```
genterm := n;
```

while the right hand side of the formula is

```
formula := n * (n + 1)/2;
```

We can use the **subs** procedure to check the basis step for the induction; here the base case is that in

$n = 1$

which .

```
subs(n = 1, genterm);
subs(n = 1, formula);
```

The results agree, so the basis step is established.

For the inductive step, we suppose the formula to be valid for $n=k$.

```
indhyp := subs(n = k, formula);
```

To sum $k+1$ terms, we compute

```
indhyp + subs(n = k + 1, genterm);
```

Finally, the formula for $n=k+1$ is

```
subs(n = k + 1, formula);
```

The results agree, so the inductive step is verified. The formula now follows by mathematical induction. Thus, you can see that, while *Maple* is not (yet) able to construct proofs entirely on its own, it is a very effective tool to use in an interactive proof construction.

Now let's consider a more complicated example. A formula for the sum

$$S=1\cdot 1!+2\cdot 2!+3\cdot 3!+\cdots+n\cdot n!$$

is much less obvious than in the preceding example. To discover one, we begin by generating some numerical data.

```
seq(sum(i * i!, i = 1..n), n = 1..20);
```

If a pattern is not immediately obvious, we can assist our intuition be generating a parallel sequence.

```
seq([n!, sum(i * i!, i = 1..n)], n = 1..10);
```

Staring at this a little renders obvious the fact that we are on to something. Let's just adjust this a little.

```
seq([(n + 1)!, sum(i * i!, i = 1..n)], n = 1..10);
```

From this evidence, we should probably infer the conjecture that a formula for our sum is

$$S=(n+1)!-1$$

The inductive proof can be carried out much as we did in the first example.

```
n := 'n': k := 'k': # clear values
S := (n + 1)! -1: genterm := n * n!:
```

The basis step is

```
subs(n = 1, genterm);
subs(n = 1, S);
```

The inductive step is

```
indhyp := subs(n = k, S); # induction step
indhyp + subs(n = k + 1, genterm);
subs(n = k + 1, S);
```

Using a little algebraic manipulation, we see that the last two formulae are equal. This completes a proof *via*mathematical induction. We conclude that our guess at the formula is correct.

3. Recursive and Iterative Definitions

Click here to access a summary of all the Maple code used in this section.

Maple functions can be defined in both procedurally (using the **proc** function) and explicitly (using the **->**notation), and each of these methods can involve recursive and iterative means of definitions. We begin our study using the **->** function of *Maple*. If we wished to define the polynomial

$$a(n)=3n^3+41n^2-3n+101$$

function , we would issue the following *Maple* command:

```
a:=n->3*n^3+41*n^2-3*n+101;
a(5);
a(523);
```

Now, if we wished to define a function recursively, say $b(n)=b(n-1)^2+2b(n-1)+6$, with the initial condition $b(0)=2$, then we would enter

```
b:=n->b(n-1)^2+2*b(n-1)+6;
b(0):=2;
b(1);
```

If we wished to see a sequence of values for the function b, we can use the **seq** function of *Maple* to display output for a given range of input.

```
seq(b(i),i=1..7);
```

Now, we shall create a similar function to **b**, called **f1**, that will find Fibonacci numbers.

```
f1 := n->f1(n-1)+f1(n-2);
f1(1) := 1;
f1(2) := 1;
f1(5);
seq(f1(i), i = 1..15);
```

While the **->** notation for functions is convenient and intuitive, it does not offer all of the facilities for improving efficiency that are available using the **proc** command. To force *Maple* to calculate these values more efficiently, we use the **remember** option to procedure definitions effected using **proc**. This option requires *Maple* to*remember* any values for the procedure that it has already computed by storing them in a table.Fibonacci

```
f2 := proc(n::integer) option remember;
  if n <= 2 then RETURN( 1 ) fi;
  f2(n-1) + f2(n-2);
end:
```

So, this procedural method encompasses both the base cases (when $n \leq 2$) and the inductive cases (as in the**else** condition). Additionally, the procedure has the `option remember` indicated, forcing *Maple* to keep track of which values of the function have already been found, so that these can be directly looked up, instead of having to be re-computed.

```
seq(f2(i), i=1..15);
```

Now, to illustrate the difference in computational complexity, we shall compare the procedural and **->** methods using the **time** function of *Maple*:

```
st:=time():seq(f1(i), i=1..20):time()-st;
st:=time():seq(f2(i), i=1..100):time()-st;
```

So, it is clear that the **remember** option can make an enormous difference in time complexity.

Another way to improve the efficiency of a recursively defined function is to rewrite it to avoid the use of recursion. Instead, we rework it so that it uses an iterative algorithm. In constructing the iterative algorithm, the key components are to create a form of loop (either a ***for*** or a ***while*** loop in *Maple*) that will compute values starting from the smallest values, upward. This method of programming is called **bottom up**: where the smallest values of a sequence are computed and then used for the larger values.

```
IterFib:=proc(n::integer)
  local x,y,z,i;
  if n=1 then y:=1;
  else x:=1; y:=1;
       for i from 2 to n-1 do
         z:=x+y;
         x:=y; y:=z;
       od;
  fi;
y;
end:
```

Contrast this with the recursive procedure **f2** which we defined earlier.

```
eval(f2);
```

Both the base cases and the recursive step are explicitly stated in the procedure body. The algorithm first attempts to compute the actual value directly, and asks for the values of sub-cases as required. This method of programming is know as "top down" for this reason: larger values are computed by breaking the input into smaller parts and combining results, similar to traversing down a binary tree.

Note that the recursive procedure with option remember and the iterative procedure perform about the same. For the first twenty Fibonacci numbers, we obtain

```
st:=time():seq(RecFib(j), j=1..100):time()-st;
```

which is quite comparable to the times we obtained for **f2**.

Note that the purely recursive implementation **f1** cannot possibly be used compute `f2(100)`. In fact, it a good exercise is to show that to do so, **f2** would need to be invoked approximately

```
f2(99);
```

times in order to handle all of the subcases which arise. Even at a billion subcases per second, this would require more than 6000 years to complete.

4. Computations and Explorations

Click here to access a summary of all the Maple code used in this section.

In this section of material, we will explore how *Maple* can be used to solve Questions 4, 5 and 8 of the*Computations and Explorations* section of the textbook.

1. How many twin primes can you find?

Solution

To determine how many twin primes there are, we will use the numtheory package of *Maple*, which contains the functions**nextprime**, **prevprime** and **ithprime**

```
restart;
with(numtheory):
list_of_primes:=[seq(ithprime(i), i=1..50)];
```

Now, having formed a list of primes, we wish to extract any twin primes that occur in this list of primes.

```
twinprime_list:=[]:
for i from 1 to nops(list_of_primes)-1 do
if list_of_primes[i+1]-list_of_primes[i]=2 then
twinprime_list
  :=[op(twinprime_list),
   [list_of_primes[i], list_of_primes[i+1]
    ]];
fi;
od;
twinprime_list;
```

Now, instead of outputting the prime pairs, the sequence number of primes may indicate a pattern. So, we will construct a list of the '**i**'s that are *twinned* together.

```
ilist:=[]:
for i from 1 to nops(list_of_primes)-1 do
   if list_of_primes[i+1]-list_of_primes[i] = 2 then
      ilist:=[op(ilist), [i, i+1]];
   fi;
od;
ilist;
```

It appears that there is no obvious pattern occurring.

2. Determine which Fibonacci numbers are divisible by 5, which are divisible by 7, and which are divisible by111. Prove that your conjectures are correct.

Solution

First, we shall generate some data to work with.

```
with(combinat): # get correct definition of 'fibonacci'
fib_list := [seq([n, fibonacci(n)], n = 0..50)]:
```

We want to determine those indices n for which the nth Fibonacci number is divisible by 5. One way to do this is to construct a list, by testing the data above, and adding to the list only those indices$n$ for which the test returns ***true***.

```
mult5 := NULL;
for u in fib_list do
  if op(2, u) mod 5 = 0 then
    mult5 := mult5, op(1, u);
  fi;
od;
mult5 := [mult5];
```

This constructs a list indicating which among the first 500 Fibonacci numbers are multiples of 5. The data indicate the the nth Fibonacci number [redacted] is divisible by 5 only if n is.

To obtain evidence for the converse, we should test whether [redacted] is divisible by 5, for as many $n$as possible. To make our test concise, and yet allow for testing a large range of values, we'll design it so that no output is produced unless a counterexample is found.

```
for n from 1 to 100 do
  if fibonacci(5 * n) mod 5 <> 0 then
    printf(`The %dth Fibonacci number %d `,
            5 * n, fibonacci(5 * n));
    printf(`(n = %d) is a counterexample`, n);
  fi;
od;
```

Hence, there is no counterexample among the first 5000 Fibonacci numbers. You can try this with values larger than 1000 also, to gain further evidence.

Another, slightly different approach can be used to locate the Fibonacci numbers divisible by a given integer, here 7. We simply build the divisibility test into the command to generate the data.

```
fib_list := seq([n, fibonacci(n) mod 7], n = 1..50);
```

We can now select the indices of those pairs whose second member is equal to 0.

```
mult7 := NULL:
for u in fib_list do
  if op(2, u) = 0 then
     mult7 := mult7, op(1, u);
  fi;
od;
mult7 := [mult7];
```

We can begin to notice a pattern in this data as follows.

```
map(x -> x / 8, mult7);
```

You can try to verify that this pattern persists by replacing 500 in the definition of **fib_list** by much larger numbers.

The tests for divisibility by 111 we leave to you.

3. The notorious $3x+1$ conjecture (also known as *Collatz' Conjecture* and by many other names) states that no matter which integer x you start with, iterating the function [redacted], where $f(x)=x/2$ if x is even and $f(x)=3x+1$ if x is odd, always produces the integer 1. Verify this conjecture for as many positive integers as possible.

Solution

To begin, we need to define the function we shall be examining.

```
Collatz := proc(n::integer)
  if type(n, even) then
    n / 2;
  else
    3 * n + 1;
  fi;
end:
```

Now we write a function that will iterate the Collatz function until the value obtained is equal to 1. We include a **count** variable for two reasons: First, we want to get some idea of how long it takes for the iterates to stabilize; second, since we don't know for certain that the iterates will stabilize for a given value of the input **seed**, we code an upper limit (large) on the number of iterates to compute.

```
IC := proc(seed::integer)
  local sentinel, count;
  count := 0;
  sentinel := seed;
  while sentinel <> 1 and count < 1000^1000 do
    sentinel := Collatz(sentinel);
    count := count + 1;
  od;
  RETURN(count);
end:
```

To verify the conjecture for the first 1000 integers, we can use our function **IC** as follows.

```
seq(IC(i), i = 1..100);
```

Note that the fact that the function eventually *stopped* is the verification that we sought.

5. Exercises/Projects

1. Use *Maple* to find and prove formulae for the sum of the first knth powers of positive integers for$n=4,5,6,7,8,9$, and 100.

2. Use *Maple* to study the McCarthy 911 function. (See Page 2277 of the text).

3. Write a *Maple* procedure to find the smallest (that is, the *first*) consecutive sequence of n composite positive integers, for an arbitrary positive integer n.

4. Use *Maple* to develop a procedure for generating Ulam numbers (defined on Page 2266 of the text). Make and numerically study conjectures about the distribution of these numbers.

5. Write a *Maple* procedure that takes an integer k as input, and determines whether or not the product of the first k primes, plus 1, is prime or not, by factoring this number.

6. Another way to show that there are infinitely many primes is to assume that there are only n primes $p_1, p_2, \ldots, p_n$. but this is a contradiction since $p_1 p_2 \cdots p_n + 1$ has at least one prime factor and it is not divisible by ■, $i = 1, 2, \ldots, n$. Find the smallest prime factor of $2 \cdot 3 \cdots p_n + 1$ for all positive integers n not exceeding 200. For which n is this number prime.

7. The *Lucas numbers* satisfy the recurrence $L_n = L_{n-1} + L_{n-2}$ and the initial conditions $L_0 = 2$ and $L_1 = 1$. Use *Maple* to gain evidence for conjectures about the divisibility of Lucas numbers by different integer divisors.

8. A sequence $a_1, a_2, a_3, \ldots$ is called *periodic* if there are positive integers N and p for which $a_n = a_{n+p}$, for all $n \geq N$. The least integer p for which this is true is called the *period* of the sequence. The sequence $a_1, a_2, a_3, \ldots$ is said to be *periodic modulo* m, for a positive integer m, if the sequence $a_1 \bmod m, a_2 \bmod m, a_3 \bmod m, \ldots$ is periodic. Use *Maple* to determine whether the Fibonacci sequence is periodic modulo m, for various integers m and, if so, find the period. Can you, by examining enough different values of m,

make any conjectures concerning the relationship between m and the period? Do the same thing for other sequences that you find interesting.

Discrete Mathematics and Its Applications

Kenneth H. Rosen, AT&T Laboratories

Chapter 5. Discrete Probability

This is a new chapter. The material for this chapter is still under development, but will be made available as soon as it is ready.

Discrete Mathematics and Its Applications

Kenneth H. Rosen, AT&T Laboratories

Chapter 6. Advanced Counting Techniques

Click here to access a summary of all the Maple code used in this section.

In this chapter we will describe how to use *Maple* to work with three important topics in counting: recurrence relations, inclusion-exclusion, and generating functions. We begin by describing how*Maple* can be used to solve recurrence relations, including the recurrence relation for the sequence of Fibonacci numbers. We then show how to solve the Tower of Hanoi puzzle and we find the number of moves required for n disks. We describe how *Maple* can be used to solve linear homogeneous recurrence relations with constant coefficients, as well as the related inhomogeneous recurrence relations. After describing how to solve these special types of recurrence relations with *Maple*, we show how to use *Maple*'s general recurrence solver. We illustrate the use of this general solver by demonstrating how to use it to solve divide and conquer recurrence relations. After studying recurrence relations, we show how to use *Maple* to help solve problems using the principle of inclusion and exclusion. Finally, we discuss how *Maple* can be used to work with generating functions, a topic covered in Appendix 3 in the text.

1. Recurrence Relations

Click here to access a summary of all the Maple code used in this section.

A recurrence relation describes a relationship that one member of a sequence [redacted] of values has to other member of the sequence which precede it. For example, the famous Fibonacci sequence [redacted] satisfies the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

Together with the initial conditions $F_1 = 1$ and $F_2 = 1$, this relation is sufficient to define the entire sequence [redacted].

In general, we can think of a recurrence relation as a relation of the form

$$r_n = f(r_{n-1}, r_{n-2}, \ldots, r_{n-k})$$

in which each term [redacted] of the sequence depends on some number k of the terms which precede it in the sequence. For example, for the Fibonacci sequence, the function f is $f(x,y) = x + y$.

To understand how we can work with recurrence relations in *Maple*, we have to stop for a moment and realize that a sequence [redacted] of *values* (numbers, matrices, circles, functions, etc.) is just a *function* whose domain happens to be the set of (usually positive) integers. If we want to take this point of view (and we do!), then the$n$th term [redacted] of a sequence [redacted] would be more conventionally written as [redacted], and we would refer to the *function*r. In this way, we can think of the sequence [redacted] as one