Chapter 4

fsqFsHn sGGousG

1 Introduction to Cryptography

If some information is meant to be kept private, the best means is to keep it well hidden. This is not, of course, always possible. One way around this is to hide the information or message in plain sight, that is, encode it in some way so that even if it is seen, it will be unreadable. The study of such encoding processes is called Cryptography[1]. The name "Cryptography" comes from the Greek words *kryptos* ($\kappa\rho\upsilon\pi\tau o\sigma$), meaning "hidden" or secret, and *graphia* ($\gamma\rho\alpha\phi\iota\alpha$), meaning writing.

Since ancient times, cryptography has been a part of military and governmental communications. More recently it has become part of nearly everyone's life because of the Internet, electronic banking, and so on.

The usual jargon is as follows: the message you want to hide is called the plaintext, and the act of encoding it is called encryption or enciphering. The encoded plaintext is called the crypttext or the ciphertext, and the act of decoding it is called decryption or deciphering (or "cracking the code", if the decoder wasn't the intended reader). Usually, an encryption system (also called a cipher) has an auxiliary piece of information called the key needed for the encoding

[1]Cryptography is related to, but distinct from, steganography, which is the process of hiding the fact that a message exists at all. Using "invisible ink" is steganography, as is hiding messages within other messages. For example, looking at the second letter of each word in the message below (which was actually sent by a German spy during World War I [Kahn, p. 521]),

> Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects pretext for embargo on byproducts, ejecting suets and vegetable oils.

one finds a rather different message, namely

> Pershing sails from NY June I.

Steganography is often used to augment cryptography, and is also related to "digital watermarking". We will not be considering steganography here.

and decoding process. Mathematically, we can represent the encryption process as

$$f_{key}(\mathcal{P}) = \mathcal{C},$$

where $\mathcal{P}$ is the plaintext and $\mathcal{C}$ is the ciphertext. To decrypt the message, one applies f^{-1} to the ciphertext. Note that to be able to decipher a message without ambiguity, f^{-1} must be a well-defined function (although f needn't be! It is perfectly all right to have the same message encode to two different ciphertexts, as long as we can get the original when deciphering.)

2 Simple Ciphers

2.1 Simple substitution

One of the most common (and very easy to crack) ciphers is substitution. One sometimes sees these in a newspaper somewhere near the crossword puzzle. This scheme is also the one used in Poe's story "The Gold Bug", and the Sherlock Holmes "Mystery of the Dancing Men". The idea is straightforward: choose a rearrangement of the letters of the alphabet, and replace each letter in the plaintext by its corresponding one.

Such substitutions are very simple to implement in Maple. We will use the `CharacterMap` function from the `StringTools` package[2] which does exactly what we want. First, we define two "alphabets"– one for our plain text, and one for our encrypted text. The second is a permutation of the first.

```
> with(StringTools):
  Alphabet :="abcdefghijklmnopqrstuvwxyz":
  Cryptabet:="THEQUICKBROWNFXJMPSVLAZYDG":
```

Now we define a pair of functions[3] to encrypt and decrypt our text.

```
> Scramble  := msg  -> CharacterMap(Alphabet, Cryptabet, msg):
  Unscramble:= code -> CharacterMap(Cryptabet,Alphabet, code):

> Scramble("please do not read this");
```

"JWUTSU QX FXV PUTQ VKBS"

[2]The `StringTools` package was introduced in Maple 7, and the string data type was introduced in Maple V release 5. Everything we do in this chapter can easily be implemented without using `StringTools`– in fact, the original version of this chapter was written for Maple Vr4 using names rather than strings. Since Maple 6 is still in widespread use (Spring 2002), we'll try to include alternatives to using this package when practical.

[3]In Maple Vr5 and Maple 6, we could define `CharacterMap` as

```
CharacterMap := (old,new,txt)->cat(seq(new[SearchText(txt[i],old)],i=1..length(txt)));
```

This will only work if all characters in the `txt` correspond to ones in `old`, so a space would have to be added to both `Alphabet` and `Cryptabet` to get this example to work.

```
> Unscramble(%);
```

"please do not read this"

Note that our message contains a spaces which are preserved in the encryption process, because the `CharacterMap` function only modifies those characters which are found in the first string. If a character isn't found, it is left alone. Thus, if we try to scramble a message containing punctuation or upper-case letters, they aren't modified by `Scramble`. Since our `Cryptabet` is all upper-case, a message which began with mixed case will be distorted.

```
> Scramble("He said I need direction; I head for the door.");
  Unscramble(%);
```

"HU STBQ I FUUQ QBPUEVBXF; I KUTQ IXP VKU QXXP."

"be said f need direction; f head for the door."

It is traditional in pen-and-paper cryptography to use a 26-letter alphabet (that is, to remove spaces and punctuation and ignore any distinction between upper and lower case letters). If you wish to do this, it could be done[4] using `LowerCase` (which translates all letters to lower case), and a combination of `Select` and `IsLower` to remove anything that isn't a letter.

```
> Select(IsLower,LowerCase("I don't mind suggestions. Don't give me any more."));
```

"idontmindsuggestionsdontgivemeanymore"

```
> Scramble(%);
  Unscramble(%);
```

"BXFVCNBFQSLCCUSVBXFSQXFVCBAUNUTFDNXPU"

"idontmindsuggestionsdontgivemeanymore"

In general we won't adhere to this convention in this chapter. Rather, we will treat spaces and punctuation like any other character. This is not a common approach, but one we happen to like.

There are two big drawbacks to such a substitution cipher. One primary problem is that the key for enciphering and deciphering is the permutation: you need to remember an ordering of the entire alphabet.

[4]How to accomplish this without `StringTools` is left to the seriously motivated reader.

A second problem, shared with all such monoalphabetic substitutions (i.e., those that substitute one letter for another), is that they aren't that hard to break. If you have a long enough message, you can use the fact that certain letters occur much more frequently than others to help tease out the original text. In English text the letter "e" is the occurs most frequently, followed by "t", "a", "n", "i", and so on. Spaces occur about twice as often as the letter "e", assuming they are encrypted, rather than just omitted. Once you guess a few letters, common words like "the" and "and" fall into place, giving more clues.

We will examine some polyalphabetic substitution ciphers in §5.

2.2 The Caesar cipher, and the ASCII encoding

A variation on the arbitrary permutation discussed above is the "Caesar cipher", named after Julius Caesar, who supposedly invented it himself. This is also what was done by a "Captain Midnight Secret Decoder Ring", popular in the 1930s and 40s. Here we convert our alphabet to numeric equivalents (with, say A=0, B=1, and so on), add an offset to each numeric equivalent (legend has it that Caesar used an offset of 3), then re-encode the numbers as letters. We do the addition modulo the length of the alphabet; that is, when we shift a Y by 3, we get B, since $24 + 3 = 27$, and $27 \equiv 1 \pmod{26}$. However, we won't always restrict ourselves to a 26 letter alphabet— we may want punctuation, numbers, and to preserve upper- and lower-case distinctions.[5]

While we could easily implement the Caesar cipher without converting our text to numbers, doing so makes it easier to understand and sets us up for more complicated ciphers we will need later.

Treating characters as numbers

Computers already have a built-in numeric representation of the alphabet— the ASCII encoding.[6] At their lowest level, digital computers operate only on numeric quantities— they just manipulate ones and zeros. The smallest piece of information is the state of a switch: it is off (0) or it is on (1). This is called a bit, which is short for **b**inary dig**it**. A group of 8 bits is called a byte, and is a single "character" of information.[7] A byte can have $2^8 = 256$ different states:

$$00000000, 00000001, 00000010, 00000011, \ldots, 11111111$$

[5]For example, the title of this chapter (fsqFsHn sGGousG) is actually the result of using a Caesar cipher on its original title. A 53-character alphabet consisting of all the upper-case letters, a space, and all the lower-case letters was used, so the space in the middle may not correspond to a space in the original. The three Gs that occur should be a good hint to decoding it.

[6]In fact, while most computers these days use ASCII, it is not the only such arrangement. IBM mainframes, for example, primarily use another encoding called EBCDIC. But this makes no real difference to our discussion.

[7]The next larger unit in a computer is a word. However, different makes of computers have different sizes of words. Most computers today have 4 byte words (32 bits), but word sizes of 2 bytes (16 bits) and 8 bytes (64 bits) are not uncommon. 8-bit bytes are now essentially universal although in the early days of computing, there were computers which used other sizes, such as 6 bit bytes.

which are generally interpreted either as representing the integers 0 through 255, or the highest bit is often taken to represent the sign, in which case we can represent integers between -128 and 127. (We get an "extra" negative number because we don't need -0, so the byte 10000000 can mean -128 instead.) For notational convenience, bytes are often written in hexadecimal (that is, in base 16), grouping the bits into two blocks of 4 bits (sometimes called a nybble, half a byte), and using the characters A–F for the integers 10–15. Thus, the byte 01011111 would be written as 5F in hexadecimal, since 0101 means $2^2 + 2^0 = 5$, and 1111 is $2^3 + 2^2 + 2^1 + 2^0 = 15$.

So how does a computer represent characters? There is an agreed upon standard encoding of characters, punctuation, and control codes into the first 127 integers. This is called ASCII,[8] an acronym for American Standard Code for Information Interchange. Extended ASCII encodes more characters into the range 128–255 (this is where you would find characters for å and ±, for example), although the characters found in this range vary a bit. Not all programs can deal successfully with characters in the extended range[9]

Maple has a built-in routine to allow us to convert characters to and from their ASCII equivalents. If it is called with a list of numbers, it returns a string of characters, and if called with a character string, out comes a list of numbers. For example,

```
>  convert("How do I work this?",bytes);

   [72, 111, 119, 32, 100, 111, 32, 73, 32, 119, 111, 114, 107, 32, 116, 104, 105, 115, 63]

>  convert([72, 117, 104, 63],bytes);

                                  "Huh?"
```

We can use this to make a list of the ASCII codes. Maple prints a □ when the ASCII code is not a printable character. We make the table for characters up through 127 because the meanings in extended ASCII differ depending on what character set you use. Feel free to make your own table of all 255 characters. (In the interests of making a nicely formatted table, we have used a some-

[8]ASCII is not the only such standard, although it is currently the most commonly used. Prior to the widespread use of personal computers in the 1980s, EBCDIC (Extended Binary Coding for Data Interchange and Communication) tended to be in more common use, because it was what IBM mainframes used. There are many other such assignments.

[9]In the late 1980s and early 1990s, there was a big push to make programs "8-bit clean", that is, to be able to handle these "unusual" characters properly. There are a number of distinct assignments of characters to the upper 128 positions. One common one is Latin-1 (ISO 8859-1), which contains the characters used in the majority of western European languages. Other assignments include characters for Scandinavian languages, Arabic, Cyrillic, Greek, Hebrew, Thai, and so on. Since the late 1990s, a multi-byte character encoding called Unicode (ISO 10646) has been in use; this universal character set allows computers to handle alphabets (such as Chinese) that need more than 255 characters. While use of Unicode is becoming more widespread, single byte character sets like ASCII will likely predominate for quite a long while.

what complicated command. The much simpler `seq([i,convert([i],bytes)],i=0..127);` would have produced the same information, just not as neatly.)

```
>   matrix([ [_,seq(i, i=0..9)],
            seq([10*j, seq(convert([i+10*j],bytes), i=0..9)], j=0..12)]);
```

```
[  _     0      1      2      3      4      5      6      7      8      9   ]
[  0    ""    "□"    "□"    "□"    "□"    "□"    "□"   "\a"   "\b"   "\t"  ]
[ 10   "\n"   "\v"   "\f"   "\r"   "□"    "□"    "□"    "□"    "□"    "□"  ]
[ 20   "□"    "□"    "□"    "□"    "□"    "□"    "□"   "\e"   "□"    "□"  ]
[ 30   "□"    "□"    " "    "!"    "\""   "#"    "$"    "%"    "&"    "'"  ]
[ 40   "("    ")"    "*"    "+"    ","    "-"    "."    "/"    "0"    "1"  ]
[ 50   "2"    "3"    "4"    "5"    "6"    "7"    "8"    "9"    ":"    ";"  ]
[ 60   "<"    "="    ">"    "?"    "@"    "A"    "B"    "C"    "D"    "E"  ]
[ 70   "F"    "G"    "H"    "I"    "J"    "K"    "L"    "M"    "N"    "O"  ]
[ 80   "P"    "Q"    "R"    "S"    "T"    "U"    "V"    "W"    "X"    "Y"  ]
[ 90   "Z"    "["    "\\"   "]"    "^"    "_"    "`"    "a"    "b"    "c"  ]
[100   "d"    "e"    "f"    "g"    "h"    "i"    "j"    "k"    "l"    "m"  ]
[110   "n"    "o"    "p"    "q"    "r"    "s"    "t"    "u"    "v"    "w"  ]
[120   "x"    "y"    "z"    "{"    "|"    "}"    "~"    "□"    "□"    "□"  ]
```

Note that some of the characters (such as ASCII code 10) print as a backslash and a letter, like \n. These are codes for special control characters: \n indicates a newline, \t is a tab, and so on. Since Maple indicates a string by printing it inside double quotes, it indicates the double quote[10] character (ASCII 34) using \", and the backslash character (ASCII 92) with a pair of backslashes. While it doesn't print out in Maple, ASCII 127 is the control sequence DEL (sometimes printed as ^?). This sometimes is interpreted as "delete the previous character"; if your output device interprets it that way, characters can seem to mysteriously vanish.

Another special character is ASCII code 0: note that it prints as nothing, not even a blank space (ASCII 32 is the space character). This is the null character, and is typically used internally to indicate the end of a string. If a null occurs in the middle of a string, usually what comes after it won't print out unless you take special steps. To see this for yourself, try a command like `convert([73,116,39,115,0,103,111,110,101,33],bytes)`. You'll see that the characters following the 0 don't print out.

The `StringTools` package also contains `Char` which gives the character corresponding to a decimal number, and `Ord` which gives the number corresponding to a character. We'll generally use `convert`, although it makes little difference.

[10]The ASCII code does not have distinct codes for the open-quote (") and close-quote (") characters used in typesetting. Neither do most computer keyboards. However, the apostrophe (') and grave accent or backquote (`) characters have distinct ASCII codes (39 and 96, respectively).

Render unto Caesar

Now we are nearly ready to write a version of the Caesar cipher. The basic idea is that for each character of the plaintext, we convert it a numerical representation, add a pre-specified offset modulo 127, then convert back to characters. Unlike the `Scramble` function we wrote in §2.1, we needn't declare an alphabet explicitly because we will use the ASCII codes. However, in addition to the Maple command `convert( ,bytes)`, we will use another new command, `map`.

The `map` command allows us to apply a function to every element of a vector or list in one call. For example, to compute the square root of each of the first 10 integers, we could do

```
> map(sqrt,[1,2,3,4,5,6,7,8,9,10]);
```

$$[1, \sqrt{2}, \sqrt{3}, 2, \sqrt{5}, \sqrt{6}, \sqrt{7}, 2\sqrt{2}, 3, \sqrt{10}]$$

If the function we wish to apply takes two variables, we can give the second one as a third argument to `map`. For example, the command

```
> map(modp,[1,2,3,4,5,6,7,8,9,10],3);
```

$$[1, 2, 0, 1, 2, 0, 1, 2, 0, 1]$$

computes $n \bmod 3$ as n ranges over the list $[1, 2, \ldots, 10]$. (Recall that $n \bmod p$ is the remainder after dividing n by p.)

We *finally* are ready. See if you can understand it before reading the explanation below. Remember to read from the middle outward. Note that we are defining the function we are `mapping` directly as the first argument.

```
> Caesar:= (plain, shift) ->
            convert(map(x -> (x+shift-1) mod 127 + 1,
                           convert(plain,bytes)),
                     bytes):
```

Let's give it a try:

```
> Caesar("Et tu, Brute?",3);
```

"Hw#wx/#EuxwhB"

If you don't see what is going on in `Caesar`, try to think of it as a composition of several functions, each of which you DO understand. That is, `Caesar` works as follows

- First, we convert the input string (called `plain`) into a list of numbers, using the call `convert(plain,bytes)`. Note that this is done on the third line, the *innermost* function.

- Next, we apply the function `x ->(x+shift-1) mod 127 + 1` to each of numbers in the resulting list using `map`. This call to `map` has two arguments: the definition of the shift, and what to apply it to (the innermost function `convert(plain,bytes)`).

- Finally, we convert the list back to characters again. This is the outermost function: it begins on the second line (immediately after the `->` and ends with the final parenthesis.

Viewing it as a composition, we have

$$\texttt{Caesar} := \texttt{convert}^{-1} \circ \texttt{map} \circ \texttt{convert}.$$

Note that we didn't really compute $(x + \texttt{shift}) \bmod 127$; instead we subtracted one from the sum, computed mod 127, then added 1 again. Why? Consider what would happen if, for example, our character was "k" (ASCII code 107) and we shifted by 20. Then $(107+20) \bmod 127$ gives 0, which is the code for the null byte, indicating the end of a string. This would be a problem. So instead, we add 1 afterward to ensure that our answer is in the range $1, \ldots, 127$. And just to avoid confusing ourselves, we subtract the 1 first, so that shifting by 0 doesn't change anything.

Aside from the fact that this function is a bit confusing to read and offers essentially no cryptographic security, what else is wrong with it? Consider the following example:

```
>   Caesar("Who put the benzedrine in Mrs. Murphy's Ovaltine?", 86);
```

".?FvGLKvK?<v9<EQ<;I@E<v@Ev$IJ□v$LIG?P}Jv&M8CK@E< □"

In addition to the fact that the result looks funny, if we were to try to decipher this from printed output, we would have a little (but not much) trouble, because the characters "." and "?", after encoding, print out as □.[11] This happens because the numeric values of these characters, after shifting by 86, are 5 and 22, respectively. Neither of those corresponds to a printing character, so we can't tell them apart. Of course, if we had no intention of dealing with our encoded message in printed form, this would be no problem. The actual character values are different, they just print out the same.

How can we get around this? We can restrict ourselves to printing characters using a variation of the idea in `Scramble` of §2.1, that is, giving an explicit `Alphabet` that we will deal in.

Before doing this, however, it will be helpful to discuss the `proc` command, which allows us to construct functions that consist of more than one Maple command.

[11] Of course, it is easy to make examples where significant parts of the message all come out as □. One such is the same message with a shift of 50.

3 Defining functions with `proc`; Local and global variables

We already know how to define functions in Maple. However, all of the functions we have defined so far have consisted of a single Maple expression (no matter how complicated it might be). For example,

```
> sqr := x -> sqrt(x);
```

$$sqr := \mathrm{sqrt}$$

```
> sqr(25);
```

$$5$$

However, Maple has a more versatile way of defining functions using the `proc` command. The function `sqr2` below behaves exactly the same as `sqr` defined above. (When using `proc`, the function begins with the keyword `proc`, followed by any parameters in parenthesis, then one or more Maple statements, and finally the keyword `end`. The result of the function is the result of the last Maple command executed.

```
> sqr2 := proc(x)
     sqrt(x);
  end;
```

$$sqr2 := \mathbf{proc}(x)\,\mathrm{sqrt}(x)\,\mathbf{end}$$

```
> sqr2(25);
```

$$5$$

```
> sqr2(-25);
```

$$5\,I$$

Unlike the functions defined by `->`, however, we can use several statements when using `proc`:

```
> sqr3 := proc(x)
    if (x >= 0) then
       sqrt(x);
    else
       print("The square root of ",x,"is not a real number");
    fi;
    end;
```

$$sqr3 := \mathbf{proc}(x)\ \mathbf{if}\, 0 \le x\, \mathbf{then}\, \mathrm{sqrt}(x)\, \mathbf{else}\, \mathrm{print}(\text{"}\textit{The square root of }\text{"}, x, \text{"}\textit{is not a real number}\text{"})\ \mathbf{fi}\ \mathbf{end}$$

```
>  sqr3(-25);
```

$$\textit{The square root of }, -25, \textit{is not a real number}$$

```
>  sqr3(25);
```

$$5$$

Here is another variation on the same idea: we compute a variable $y = x^2$. If $y > 25$, the result is 25, otherwise it is y. [12]

```
>  mesa:= proc(x)
     y := x^2;
     if ( y>25) then
        25;
     else
        y;
     fi;
   end;
```

```
Warning, 'y' is implicitly declared local
```

$$mesa := \mathbf{proc}(x)\ \mathbf{local}\, y;\ y := x^2;\ \mathbf{if}\, 25 < y\, \mathbf{then}\, 25\, \mathbf{else}\, y\ \mathbf{fi}\ \mathbf{end}$$

What does that warning mean? It means that Maple has assumed that the variable `y` used in `mesa` is `local` to that procedure. That is, it is different from a variable `y` that may be used outside the context of the procedure. Let's check that. First, we set `y` to 35, invoke `mesa(19)` (which sets *its* copy of `y` to 361), then check what the value of `y` is.

```
>  y:=35;
```

$$y := 35$$

[12]Note that this could also be done using `piecewise`.

```
> mesa(19);
```

$$25$$

```
> y;
```

$$35$$

It is still 35. Now, let's add a line saying that `y` has a *global* definition— the `y` within the procedure is the same as the `y` outside.

```
> mesa2:= proc(x)
    global y;

    y := x^2;
    if ( y>25) then
       25;
    else
       y;
    fi;
  end;
```

$$mesa2 := \mathbf{proc}(x)\,\mathbf{global}\,y;\; y := x^2\,;\; \mathbf{if}\,25 < y\,\mathbf{then}\,25\,\mathbf{else}\,y\,\mathbf{fi}\;\mathbf{end}$$

```
> y;
```

$$35$$

```
> mesa2(19);
```

$$25$$

```
> y;
```

$$361$$

This time, of course, `y` was indeed changed by invoking `mesa2`.

The opposite of `global` is, not surprisingly, `local`. In order to stop Maple from giving warnings about variables being implicitly declared local, we can add a the statement `local y;` just after the `proc` statement. Using such statements is called declaring the scope of the variables. This is a good habit to get into, because it lessens the chance of accidentally using a global variable or of misspelling the name of a variable. Some computer languages require that you declare all variables you use.

Another good habit to get into is to indicate what type of arguments the function will accept. This is optional in Maple, and not always desirable (you may not always know what they will be). This is done by specifying the type after two colons in the argument list. For example,

```
>  ith:=proc(l::list, i::posint)
     return(l[i]);
   end:
```

is a function that insists its first argument be a list, and the second must be a positive integer. If you call it with something else, you will get an error message.

```
>  ith("Henry",8);

Error, invalid input: ith expects its 1st argument, l, to be of type list, but
received Henry
```

4 Caesar cipher redux

Now that we know about `proc`, let's implement the Caesar cipher again, this time with an alphabet of our choosing.

First, we define a global variable `Alphabet` that lists all the characters we intend to encode. We would like to be able to distinguish all characters which print out, as well as blanks (that is, those which you have keys for on your keyboard). To avoid typing in the full list of characters we want[13], we generate a list of all the integers less than 256 and convert it to a character string with `convert(,bytes)`. Then we use `Select` and `IsPrintable` to pick out the usual printable characters. Finally, since we would like to have a space and a newline as characters in our alphabet, we add them to the resulting string with `cat`.

```
>  Alphabet := cat("\n\t", Select(IsPrintable,convert([seq(i,i=1..255)],bytes)));
```

```
"\n\t !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWX\
YZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~"
```

We will use this particular alphabet regularly in the next few sections.

Next, we define two functions which perform the analogue of `convert(,bytes)`, but with our special alphabet. `ListToString` will take a list of integers and replace it with the character from that position in `Alphabet`, and `StringToList` does the reverse process. For each character in the input string, `StringToList` uses `SearchText` to find its position in `Alphabet`. We need

[13]For versions prior to Maple 7, just type in the desired alphabet directly

to subtract one from this position because we would like our characters to be numbered from 0 rather than 1. Thus, in the `Alphabet` above, \t is character 0, and the numeric code for ∼ is 96. `Alphabet` is declared as a global variable.

```
> StringToList := proc(text::string)
    local i;
    global Alphabet;
    [seq(SearchText(text[i],Alphabet)-1, i=1..length(text))];
  end:
```

```
> ListToString := proc(numlist::list(nonnegint))
    local i;
    global Alphabet;
    cat(seq(Alphabet[numlist[i]+1], i=1..nops(numlist)));
  end:
```

Now the Caesar cipher is written in terms of these functions. Note that since 0 corresponds to a printable character, we don't have to add and subtract 1 as before.

```
> Caesar2:= proc(plaintext::string, shift::integer)
    local textnum,codenum,i,p;
    global Alphabet;

    p       := length(Alphabet);
    textnum := StringToList(plaintext);
    codenum := [seq( modp(textnum[i]+shift, p), i=1..length(plaintext)) ];
    ListToString(codenum);
  end:
```

It works as follows.

- We are given a `plaintext` message, and a shift amount. The temporary variables that we will use during the conversion process are declared as local, and the `Alphabet` is declared as global.
- The number of characters in the alphabet is saved as `p`.
- The plaintext is converted to a list of numbers called `textnum`.
- Each number in `textnum` is shifted, and the result is taken mod `p`. Note that since `StringToList` yields result in $\{0, 1, \ldots, p-1\}$, we don't have to monkey around to get the result into the proper range like we did in the first `Caesar` (§2.2, p. **4**:7). Also note that we use `modp(a,p)` to compute $a \bmod p$, instead of writing `a mod p` as before. Either way works fine.
- Finally, the result is converted back to characters with `ListToString`.

Let's try it out.

```
> Caesar2("Veni, Vidi, Vici",3);
```

“Yhql/#Ylgl/#Ylfl”

To decode, we can just use the negative of the shift amount.

```
> Caesar2(%,-3);
```

“Veni, Vidi, Vici”

```
> text:="I have heard the mermaids singing, each to each.
  I do not think that they will sing to me.";
```

text := “I have heard the mermaids singing, each to each.\nI do not think \
that they will sing to me.”

In this sample, we have used a text which contains a newline character. Maple represents this as \n inside a text string. Since we will treat the newline as a regular character, it too will encode to something. If you look carefully at the output below, you can probably pick it out.

```
> Caesar2(text,64);
```

“(’G@UD’GD@QC’SGD’LDQL@HCR’RHMFHMFl’D@BG’SN’D@BGnˆ\
(’CN’MNS’SGHMJ’SG@S’SGDX’VHKK’RHMF’SN’LDn”

```
> Caesar2(%,-64);
```

“I have heard the mermaids singing, each to each. \nI do not think that they \
will sing to me.”

If we would like to see the decrypted text with the newlines expanded, we need to use `printf`.

```
> printf(%);

I have heard the mermaids singing, each to each.
I do not think that they will sing to me.
```

One thing that we should check is what happens if our message should contain a character that is not in our alphabet. Rather than changing our text, we'll dramatically shorten the alphabet.

```
> Alphabet:="abcdefghijklmnop_";
```

Alphabet := “abcdefghijklmnop_”

Now let us go then, you and I, and re-encode the test message using the new alphabet, but with a shift of zero. Ordinarily, a shift by 0 would not change the text at all. But with the shortened alphabet, something happens to the characters that aren't mentioned.

```
> Caesar2(text,0);
```

"__ha_e_hea_d__he_me_maid___inging__each__o_each____do_no___hink__ha___he___ill__ing__o_me_"

Can you explain why this happens?

5 Improved Caesar-like ciphers

Certainly the Caesar cipher offers no cryptographic security at all: if you know the alphabet the message was encoded in, you need only guess one character to crack the code. Even if you don't know the alphabet, guessing the correspondence is not very hard with a little patience.

In this section, we will discuss a few approaches to improving the security, while retaining the basic idea of character shifting.

5.1 The Vignère cipher

One way to make a Caesar cipher a bit harder to break is to use different shifts at different positions in the message. For example, we could shift the first character by 25, the second by 14, the third by 17, and the fourth by 10. Then we repeat the pattern, shifting the fifth character by 25, the sixth by 14, and so on, until we run out of characters in the plaintext. Such a scheme is called a Vignère cipher[14], which was first used around 1600, and was popularly believed to be unbreakable.[15] This cipher is called a polyalphabetic substitution cipher, because several different substitutions are made depending on the position of the character within the text.

In our first example, the key consists of the four shifts $[25, 14, 17, 10]$, which are the numerical equivalents of the string "ZORK" in a 26-letter alphabet consisting of the letters A–Z. It is common practice to think of our key as plaintext letters, rather than their numerical equivalents, but either will do. We can encode the string "CRYPTOGRAPH" as

$$\begin{array}{ccccccccccc} C & R & Y & P & T & O & G & R & A & P & H \\ +25 & +14 & +17 & +10 & +25 & +14 & +17 & +10 & +25 & +14 & +17 \\ B & F & P & Z & S & C & X & B & Z & D & Y \end{array}$$

Note that in the above, the letter "R" in the plaintext encodes to both "F" and "B" in the crypttext, depending on its position. Similarly, the two "Z"s in the crypttext come from different plain characters.

[14]This cipher takes its name after Blaise de Vignère, although it is actually a corruption of the one he introduced in 1585. Vignère's original cipher changed the shift amount each letter based on the result of the last encoding, and never repeated. This scheme is *much* harder to break. However, one reason for its lack of popularity was probably due to the fact that a single error renders the rest of the message undecipherable. More details van be found in [Kahn].

[15]In fact, as late as 1917, this cipher was described as "impossible of translation" in a respected journal (Scientific American), even though the means to break it had been well known among cryptographers for at least 50 years.

Now we implement this in Maple. This is very similar to the Caesar cipher, just with the extra complication of multiple shifts, and letting our key be a string.

First, we set our Alphabet to the usual one. We also use the conversion functions from §4.

```
> Alphabet := cat("\n\t", Select(IsPrintable,convert([seq(i,i=1..255)],bytes))):
```

```
> Vignere:= proc(plaintext::string, key::string)
    local textnum,codenum,i,p,offsets,keylen;
    global Alphabet;

    p       := length(Alphabet);
    offsets := StringToList(key);
    keylen  := length(key);
    textnum := StringToList(plaintext);
    codenum := [seq( modp(textnum[i] + offsets[modp(i-1,keylen)+1], p),
                     i=1..length(plaintext)) ];
    ListToString(codenum);
  end:
```

To try it out, we'll the same text as in the previous section. Notice how much harder it is to pick out the word boundaries in the resulting ciphertext.

```
> coded:=Vignere(text,"Prufrock");
```

```
coded := "{t^HiUeT6ThKtdLQR`[Y`QMPDtiPaWMZ8\twLTSLmEbwLTSL \
{Rr?hW_eZ@gw[[YRWRg^HgqXT6lw^\\\\PmD\\dNtdSm>X$"
```

We can make the decoding function from the original (let's call it `unVignere`) by changing exactly one + sign to a −.[16] We omit the change here so perhaps you will figure it out for yourself. But we will test it, to show you that it does work.

```
> printf(unVignere(coded,"Prufrock"));

I have heard the mermaids singing, each to each.
I do not think that they will sing to me.
```

Even though this scheme looks quite daunting, it is not so very hard to crack if you use a computer or have a very large supply of perseverance. If we know that the key is of a certain length, say 4, and our plaintext is sufficiently long, then we can perform frequency analysis on every fourth letter. Even if we don't know the key length, it is not too hard to write a computer program to try all the lengths less than, say, 10, and pick the one that looks best.

5.2 One-time pads

Note that the longer the key is in the Vignère cipher, the harder it is to break. If the key were as longer than the text, then it might seem at first that analyzing the frequency of letters in the encrypted text would be of no help, since each letter would be shifted by a different

[16]Note that we could also use the `Vignere` routine, but with the inverse of the key. For example, in the preceding example, the inverse of "`Prufrock`" is "`M+(7+.:2`": the numeric code of `P` plus the numeric code of `M` is 97 (the length of the alphabet), similarly for `r` and `+`, `u` and `(`, and so on.

amount. This is *almost* true. If the key is an passage of text in English, then the shifts will occur with a predictable frequency. Of course, the problem gets very difficult, but cryptanalysts are persistent people.

But what if there were no predictability within the key, having the shifts come at random? The result (a Vignère cipher with an infinitely long, completely random key) is an cryptosystem that *cannot be broken*. Since the shifts are random, if you manage to decipher part of the message, this gives you no clue about the rest. Furthermore, any plaintext of a given length can encrypt to any ciphertext (with a different key, of course). For example, the ciphertext "=5nwhn KDNO?uWTBC-XA" might have come from the phrase "Let's have dinner." (with the key "pOyntmbbXYtrjSTGe1"), or it might be the encryption of "Attack at midnight" with the key "{@y4#!Jbz>&moSYEoL". Since any message could encrypt to any other, there is no way to break such a code unless you know the key.

But that is the problem: the key is infinitely long. Infinitely long, truly random sequences of numbers tend to be somewhat unwieldy. And to decode the message, you must know what random sequence the message was encoded with.

Such a system is called a one-time pad, and was used regularly by spies and military personnel. Agents were furnished with codebooks containing pages and pages of random characters. Then the key to the encryption is given by the page on which to begin. It is, of course, important that each page be used only once (hence the name "one-time pad"), because otherwise if a codebreaker were able to intercept a message and (via some other covert means) its corresponding translation, that could be used to decipher messages encoded with the same page. This sort of setup makes sense if an agent in the field is communicating with central command (but not with each other). Each agent could be given his own codebook (so that if he is captured, the whole system is not compromised), and he uses one page per message. Central command has on file the books for each agent.

A variation on this theme is the Augustus cipher,[17] where instead of a random sequence of shifts, a phrase or passage from a text which is as long as the plaintext is used. The trouble with this is that, because of the regularities in the key, a statistical analysis of the crypttext allows one to break the cipher.

Another issue is that to be truly unbreakable, the random sequence must be truly random, with no correlation among the characters. This is harder than it sounds— real randomness is hard to come by. If the random sequence has some predictability, the resulting stream can be attacked. A number of attacks on cryptosystems have been made not by breaking the encryption scheme directly, but because the underlying random-number generator was predictable.

We can easily modify our `Vignere` program to be a one-time pad system, using Maple's random number generator to make our one-time pad.[18] You might think that generating a

[17]Sometimes a Caesar cipher with a shift of +1 is also called an "Augustus Cipher", even though these are very different ciphers.

[18]Technically speaking, this is not a one-time pad, but a one-time stream. The distinction is subtle, and we will ignore it here.

random sequence of numbers would be inherently unreproducible, so the message would be indecipherable unless we record the stream of random numbers. However, computers can not usually generate a truly random sequence. Instead, they generate a pseudo-random sequence $s_1, s_2, s_3, \ldots$ where the pattern of the numbers s_i is supposed to be unpredictable, no matter how many of the values of the sequence you already know. However, to start the sequence off, the pseudo-random number generator requires a seed— whenever it is started with the same seed, the same sequence results. We can use this to our advantage, taking the seed to be our key.[19] Note that in order to decode the message by knowing the key (the seed), the recipient must use the *same* pseudo-random number generator.

Maple's random number generator gives different results when called in different ways. If called as `rand()`, it gives a pseudo-random, non-negative 12 digit integer. When called as `rand(p)`, it gives a procedure that can be called later to generate a random integer between 0 and `p`; it is this second version that we will use. The seed for Maple's random number generators is the global variable `_seed`.[20]

```
>  OneTimePad := proc(plaintext::string, keynum::posint)
     local textnum,codenum,i,p,randnum;
     global Alphabet,_seed;

     p       := length(Alphabet);
     randnum := rand(p);
     _seed   := keynum;
     textnum := StringToList(plaintext);
     codenum := [seq((textnum[i] + randnum()) mod p, i=1..length(plaintext))];
     ListToString(codenum);
   end:
```

In this implementation, it is assumed that the key is a positive integer (it can be as large as you like). It would be easy to change it to use a string of characters, however, by converting the string to a number first. One way to do that is discussed in the next section.

In most descriptions of one-time pad systems, one takes the exclusive-or (XOR) of the random number and the integer of the plaintext, rather than adding them as we have done. This does not significantly change the algorithm if the random sequence is truly random, since adding a random number is the same as XOR'ing a different random number. However, the XOR approach has the advantage that the enciphering transformation is its own inverse. That is, if we produce the crypttext using `crypt:=OneTimePadXOR(plain,key)`, then `OneTimePadXOR(crypt,key)` will give the decryption with the same key. This is not the case

[19]Pseudo-random number generators appropriate for cryptography are rare. Most implementations (including Maple's) are good enough for everyday use, but not enough to be cryptographically secure. By analyzing the output of a typical random number generator, a good cryptanalyst can usually determine the pattern. For example, Maple's `rand` function (and that of most computer languages, such as C, Fortran, and Java) gives the result of a affine sequence of numbers, reduced to some modular base. That is, $x_i = ax_{i-1} + b$ and $s_i = x_i \bmod n$ for some fixed choices of a, b, and n. In this setting, the seed is x_0. We shall ignore the problem that this sequence is guessable, but if you want real security, you cannot.

[20]We can choose a "random" seed (based on the computer's clock) using the function `randomize()`.

for the version given above; to make a decryption procedure, we would need to modify the above by changing the + to a −.

5.3 Multi-character alphabets

We can also improve security a bit by treating larger chunks of text as the characters of our message. For example, if we start with the usual 26-letter alphabet A–Z, we can turn it into a 676-letter alphabet by treating pairs of letters as a unit (such pairs are called digraphs), or a 26^3-letter alphabet by using trigraphs, or triples of letters. This makes frequency analysis much harder, and is quite easy to combine with the other crytptosystems already discussed. We will use 99-graphs on a 256-letter alphabet (the ASCII code) when we implement the RSA cryptosystem in §11.2. While frequency analysis is still possible (charts of digraph frequencies are readily available, trigraphs less so), the analysis is much more complex.

To convert the digraph "HI" to an integer (using a length 26^2 alphabet of digraphs), one simple way is to just treat it as a base-26 number. That is, "HI" becomes $7 \times 26 + 8 = 190$, assuming the correspondence of H=7, I=8. To convert back, we look at the quotient and remainder when dividing by 26. For example, $300 = 26 \times 11 + 14$, yielding "LO".

Either we can do this arithmetic ourselves directly, or we can use the `convert(,base)` command. This command takes a list of numbers in one base and converts it to another. One slightly confusing fact is that in this conversion, the least significant figure comes first in the list,[21] instead of the usual method of writing numbers with the most significant digit first.

For example, $128 = 1 \times 10^2 + 2 \times 10^1 + 8 \times 10^0$ would be written as `[8,2,1]`. To convert this to base 16, we would note that $128 = 8 \times 16^1 + 0 \times 16^0$, so in base 16, it is written as 80.

Doing this calculation in Maple, we have

```
> convert([8,2,1], base, 10, 16);
```

$$[0, 8]$$

Below is one way to implement the conversion of text to numeric values for k-graphs. We assume our usual functions `StringToList` and `ListToString` are defined (see §4), as well as the global `Alphabet`. The routine below converts `text` into a list of integers, treating each block of `k` letters as a unit. A block of `k` characters $c_1c_2c_3 \ldots c_k$ is assigned the numeric value $\sum_{i=0}^{k-1} x_i p^k$, where x_i is the numeric equivalent of c_{i+1} assigned by `StringToList`.

[21]Such a representation is called little-endian, as opposed to a big-endian one. Some computers represent numbers internally in big-endian format (Sun SPARC, Power Macintosh), and others use a little-endian representation (those using Intel processors, which is what MS-Windows and most versions of Linux run on). This name comes from Gulliver's Travels, where in the land of *Blefuscu* there is war between the big-endians (who eat their eggs big end first), and the little-endians (who eat eggs starting from the little end).

```
> StringToKgraph := proc(text::string, k::posint)
     local p;
     global Alphabet;
     p:= length(Alphabet);
     convert(StringToList(text), base, p, p^k);
  end:
```

```
> KgraphToString := proc(numlist::list(nonnegint), k::posint)
     local p;
     global Alphabet;
     p:=length(Alphabet);
     ListToString( convert(numlist, base, p^k, p));
  end:
```

In the examples below, we are using our usual 97-character alphabet. Of course, this will work on any alphabet, although the specific numbers will differ.

```
> StringToKgraph("two by two",2);
```

```
[8719, 275, 8895, 8344, 7946]
```

In our alphabet, "t" is character number 86 and "w" is number 89, so the digraph "tw" encodes as $86+89\times97 = 8719$ (remember we are using a little-endian representation). Similarly, "o " gives $2 \times 97 + 81$, and so on. Notice that the two occurrences of "two" give different numbers, because one begins on an even character and the other starts on an odd one. The first corresponds to the digraphs "tw" and "o ", while the second is " t" and "wo".

We can also encode with 4-graphs, if we like.

```
> StringToKgraph("two by two",4);
```

One advantage (for us) of Maple's use of the little-endian order is that we needn't worry whether the length of our text is divisible by k. In the above example, the last 4-graph is "wo", which is encoded as though it had two more characters on the end with numeric code of 0. The disadvantage of this is that if our text ends with the character with code 0 (in our standard 97-character alphabet, this is a newline), that character will be lost.

Another way to treat multiple characters together is to think of them as vectors. For example, the digraph "by" might correspond to the vector [68, 91]. We will treat this approach in §8.

6 Reading and Writing from a file

While none of the methods we have covered yet are secure enough to foil a good cryptanalyst, the `OneTimePad` procedure of §5.2 is good enough for household use (unless you live with cryptanalysts!) Should we want to really use it, however, we would find it extremely tedious to type in our message each time, and then copy the encrypted text from the worksheet.

Fortunately, Maple is able to read and write data from a file, using `readline` and `writeline`. These read (or write) a single line of data from a specified file.[22]. When reading from a file, `readline` returns either a line of data, or 0 when the end of the file is reached.

Although it is not strictly necessary, it is a good idea to explicitly `open` the file before dealing with it, and `close` it when you are done. In addition to being good practice (like explicitly declaring variables), it also allows you to specify certain options, like whether you intend to read or write to the file. In fact, I recommend using `fopen`, which also allows you to specify whether the file should be treated as text or binary.[23] Below is a small example of a procedure which encrypts a file one line at a time using a procedure `EncryptLine`. We have never given such a procedure, but it should be clear how to modify it to use your favorite encryption routine instead.

```
> Cryptfile := proc(plainfilename,cryptfilename,key)
    local line, cline, file, cryptf;

    file  :=fopen(plainfilename,READ,TEXT);
    cryptf:=fopen(cryptfilename,WRITE,TEXT);

    line :=readline(file);
    while (line <> 0) do
       writeline(cryptf,EncryptLine(line,key));
       line :=readline(file);
    od;
    close(file);
    close(cryptf);
  end:
```

7 Affine enciphering

We have just examined a trivial encryption scheme, the Caesar cipher, which can be described as applying $x \mapsto x + b \pmod{p}$ to each of the characters x in a message. As we have seen, this provides no security at all (a message encoded in this way can be cracked in a short while by a third grader), but minor modifications such as those in §5 can greatly improve the security.

We now return to trivialities, but make it one step harder. Instead of encoding each character by just adding a constant, we can multiply by a constant as well. That is, we can use $x \mapsto$

[22]If you are treating a newline as a character, you will need to deal with the file on a byte-by-byte basis, using `readbytes` and `writebytes`

[23]On several operating systems, including MS-Windows and MacOS (but not Unix or Linux), files may be treated as containing text or binary data. When dealing with a text file, some control characters (such as the end-of-line marker) are translated to a format more suitable for use. When opening the file, we can tell Maple whether it should be dealt with as text or binary. If you neglect to specify which, you may find your file has one very long line if you open it in a text editor. Under Unix, text and binary files are the same, so no translation is necessary, but there is no harm in specifying the file type.

$ax + b \pmod p$ to encode each character. This not only helps a tiny bit (to crack it, you need to find out *two* characters instead of just one, and the encoded message looks a little more "mixed up", since letters adjacent in the alphabet no longer encode to adjacent letters), it helps us set up a little more conceptual machinery.

7.1 When do affine encodings fail?

You might be wondering if this will work at all. Certainly when we were just shifting the letters by a fixed amount, there was no chance that the enciphering transformation f could fail to be invertible. Does the same hold true if we multiply the numeric codes for the letters? That is, do different letters always encode to different letters?

The answer is, unfortunately, not always. For example, suppose we use a 27 character alphabet (A-Z and blank). Then if we encode using the transformation $x \mapsto 6x + 2 \pmod{27}$, both "A" (0) and "J" (9) encode to the character "C" (2). What went wrong? Recall that when we work modulo 27, any numbers that differ by a multiple of 27 are equivalent. Since 27 has nontrivial factors, several characters will collide whenever the multiplicand (6) shares a common divisor with 27. This will never happen if $\gcd(a, p) = 1$.

Let's make that precise:

Proposition 7.1 *Let a, b, x, and y be integers in the set $\{0, 1, \ldots, p-1\}$, with $\gcd(a, p) = 1$. Then*

$$ax + b \equiv ay + b \ (mod p) \qquad \Longleftrightarrow \qquad x \equiv y \ (mod p)$$

Proof. Suppose $ax + b \equiv ay + b \pmod p$. We want to show that $x \equiv y \pmod p$. Certainly we have $ax \equiv ay \pmod p$, so there must be integers k and m so that $kp + ax = mp + ay$. But we can then rewrite this as

$$a(x - y) = (k - m)p$$

that is, $a(x - y)$ is a multiple of p. If a and p share no common divisors, then $x - y$ must be a multiple of p, so $x \equiv y \pmod p$.

If $x \equiv y \pmod p$, then certainly $ax + b \equiv ay + b \pmod p$. □

Thus, by the above, we can see that the affine encoding scheme will be just fine if the multiplicand a is relatively prime to the length of our alphabet (p). Henceforth, we will always take an alphabet whose length is a prime number, so we need not worry about this issue. An alphabet of length 97 works quite well, because 97 is prime and large enough so that we can include all the usual printable ASCII characters (see §2.2, especially the table on page **4**:6). If you plan to use the full ASCII or extended ASCII character set (including the null character), then you will need to make sure you choose a to be an odd number (since the length of the alphabet will be either 128 or 256, both or which are powers of 2).

7.2 Implementing and using an affine encoding

Implementing this scheme is a trivial modification to the `Caesar2` procedure we wrote in §4. We only need add another parameter `a`, and change the encoding calculation. We also renamed `shift` as `b` to correspond to our discussion here.

In light of the Prop. 7.1, we will also add some error checking to ensure that `a` and the length of the alphabet are relatively prime. Recall that if they are not, the encryption will produce a message that cannot be decrypted. We use the `error` command[24] if this is the case. While we typically use an alphabet of prime length, we may not always do so.

```
> AffineCode:= proc(plaintext::string, a::integer, b::integer)
    local textnum,codenum,i,p;
    global Alphabet;

    p       := length(Alphabet);
    if (gcd(a,p)>1) then
       error("The %-1 parameter %2 must be prime to the length of Alphabet %3",
             2,a,p);
    fi;
    textnum := StringToList(plaintext);
    codenum := [seq( modp(a*textnum[i]+b, p), i=1..length(plaintext)) ];
    ListToString(codenum);
  end:
```

```
> mess:=AffineCode("A fine mess, Stanley!",10,5);
```

mess := “ˆ7Lj;B71BmmN7Pw{;’BHA”

What about decoding a message encoded by an affine cipher? If we know the original `a` and `b`, we could either write `AffineDecode`, which calculates `modp( (textnum[i]-b)/a, p)` instead, or, we could use `AffineCode(1/a, -b/a)`, which is really the same thing. We'll take the latter approach.

If you think about it for a second, you may wonder whether $1/a \bmod p$ makes any sense at all. It does, but only if we interpret division in the right way. We should not divide by a in the "ordinary way" and then reduce modulo p, but instead interpret it to mean the solution x to the equation $ax \equiv 1 \pmod{p}$. That is, $1/a \bmod p$ is the multiplicative inverse of a in $\{0, 1, \ldots, p-1\}$. Such an element exists exactly when a and p are relatively prime, as a consequence of Prop. 7.1. Since Maple knows about division modulo p, it allows us to use this notation, which is quite convenient.

```
> 1/6 mod 97;
```

In our definition of `AffineCode` we insisted that `a` and `b` be integers. This means that using a call like `AffineCode(mess,1/10,-5/10)` will give an error, since `1/10` is not an integer.

[24]Versions prior to Maple 6 need to use `ERROR` instead, which has slightly different behavior.

We can fix that by changing the definition of `AffineCode` to be `proc(plaintext::string, a::rational, b::rational)` or we can explicitly calculate these modular expressions before passing them to `AffineCode`.

```
> AffineDecode:= proc(ciphertext::string, a::integer, b::integer)
     local A, B, p;
     global Alphabet;
     p := length(Alphabet);
     A :=  1/a  mod p;
     B := -b/a  mod p;
     AffineCode(ciphertext, A, B);
  end:
```

```
> AffineDecode(mess,10,5);
```

"A fine mess, Stanley!"

7.3 Breaking an affine cipher

Suppose we have a message that was encrypted with an affine cipher, and we also know the `Alphabet` in use. Suppose also that we know (or manage to guess) the encoding of two letters. How do we then decipher the message? It is quite straightforward: since the message was enciphered with $x \mapsto y = (ax + b) \bmod p$, and we have two examples of x and y, we need only solve a pair of linear equations $\bmod p$. This is quite straightforward to do by hand, but Maple will also do it for us.

For example, suppose we receive the message

nrIbQyxEbkrI bwr EUbJaPybL abpyJJy UbUlrrxU

which we know is encoded in the 53 letter alphabet consisting of the letters A–Z, a blank, then a–z. Suppose we also guess that the letter "b" (28) is the encoding for a space (26), and that "U" (20) is an encoded "s" (45) . We can figure out the encoding transformation by solving the pair of equations

$$26a + b \equiv 28 \pmod{53} \qquad 45a + b \equiv 20 \pmod{53}.$$

We have Maple do this for us using `msolve`.

```
> msolve({a*26 + b = 28, a*45 +b = 20},53);
```

$$\{b = 25, a = 47\}$$

Thus, we see that the message was encoded by $x \mapsto 47x + 25 \pmod{53}$, and can be decrypted using $a = 1/47 \pmod{53}$ and $b = -25/47 \pmod{53}$. Of course, it would have been more efficient to solve for the *deciphering* transformation directly.

```
> msolve({A*28 + B = 26, A*20 +B = 45},53);
```

$$\{A = 44, B = 13\}$$

```
> Alphabet:="ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz":
  AffineCode("nrIbQyxEbkrI bwr EUbJaPybL abpyJJy UbUlrrxU", 44, 13);
```

"You bend your words like Uri Gellers spoons"

8 Enciphering matrices

8.1 Treating text as vectors

We can let our enciphering transformation f operate on larger blocks of text than just a single character at a time. However, instead of treating a multi-byte string as a "super-character", as in §5.3, we could think of it as a vector of characters. Thus, if we were using, say, 2-vectors, then the name "Dougal" would become the three vectors [68, 111], [117, 103] and [97, 108] (assuming we are using the ASCII codes). If the length of the text we were converting was not divisible by 2, we would have to extend it by adding a character to the end. Typically, this would be a space, or perhaps a null character (ASCII 0). Thus the name "Ian" occurring at the end of a message would become "Ian ", and might encode as the pair of vectors [73, 97] and [110, 32].

Below is an implementation of this conversion, `StringToVects`, which takes as input the text to convert to n-vectors and the length of the vectors, n. It uses the `StringToList` procedure from §4, as well as the usual Maple commands.

```
> StringToVects:= proc(text::string,n::posint)
    local i,j,textnums,vectors;
    textnums := StringToList(cat(text, seq(" ", i=1..modp(n-length(text),n))));
    [seq([seq(textnums[(j-1)*n + i], i=1..n)],
          j=1..nops(textnums)/n)];
  end:
```

The first thing it does is extend the length of the text to a multiple of `n`, if needed. This relies on the observation that $k + ((n - k) \bmod n)$ is a multiple of n, and on the fact that Maple will do the right thing with a command like `seq(i,i=1..0)`, that is, produce nothing whatsoever. Then the text, extended by the correct number of blanks, is converted to a list of numbers.

We now parcel up the list of numbers `textnums` into chunks of size `n`. This is done by noting that the characters in the `j`-th vector will come from positions $n(j-1)+1$ through $n(j-1)+n$, so the inner `seq` gives us the `j`-th vector, as the outer one ranges over all the vectors. If you don't see how this works right off, just take a moment to write out an explicit example for yourself.

Strictly speaking, the result is a list of lists, rather than a list of vectors. Maple is happy to accept a list in contexts where a vector is wanted.

Converting back from a list of n-vectors to a character string is easy, because we already have `ListToString` to convert a list of numbers to a character string for us. All that is needed is remove the inner structure from each of our vectors, leaving only the sequence of numbers. We can use `map` and `op` to do this for us.

```
> map(op,[[36, 79], [85, 71], [65, 76]]);
```

$$[36, 79, 85, 71, 65, 76]$$

One small problem is that for Maple, there is a difference between a vector and a list of numbers— a vector has a little more structure. While we can use the two almost interchangeably (in the same way that we can think of a point in $\mathbb{R}^n$ as an n-vector, or of as an n-vector as a point), some functions such as `op` that deal with the internal structure will give different results.

```
> l := [vector([36,79]), vector([85, 71]), vector([65, 76])]:
  map(op,l);
```

$$[1..2, [1 = 36, 2 = 79], 1..2, [1 = 85, 2 = 71], 1..2, [1 = 65, 2 = 76]]$$

In order to be sure that our function will work with both vectors and lists, we need to convert the vectors to lists first. This is easily done with the Swiss army knife `convert`. The unusual-looking type declaration in the function says that we will be happy to accept a list of lists, a list of vectors, or a list of Vectors,[25] provided their elements are integers.

```
> VectsToString:=
    proc(vects::list({list(integer), vector(integer), Vector(integer)}))
      local l;
      l:=map(convert, vects, list);
      ListToString(map(op, l)):
  end:
```

8.2 Affine encoding with matrices

Now that we know how to think of character strings as vectors, we can manipulate them. Our enciphering transformation is just a permutation of the n-vectors.

For example, the Vignère cipher (§5.1) can be interpreted as the applying the map $v \mapsto v + b \bmod p$, where v and b are vectors the length of the key.

[25]Maple has two different linear algebra packages, `linalg` and the newer `LinearAlgebra` introduced in Maple 6. The `vector` type is from the former, `Vector` from the latter.

We can do a more general trick using the affine transformation

$$v \mapsto Av + b \bmod p$$

where A is an $n \times n$ matrix, and b and v are n-vectors. As before, we need certain restrictions on the matrix A. In particular, we need it to be an invertible matrix (so that we can decipher the encoding). This is guaranteed by requiring that the determinant of A be nonzero and share no common factors with p. As before, choosing an alphabet of prime length makes our lives much easier. The linear version of this encoding, $v \mapsto Av$, is sometimes called a Hill cipher after Lester Hill, who studied such cryptosystems in the late 1920s and early 1930s [Hil1], [Hil2].

In Maple, one does matrix arithmetic pretty much as you would expect, except that we need to use `&*` to indicate matrix multiplication. This is because matrix multiplication is not commutative, while `*` is. Using `&*` tells Maple not to attempt the multiplication right away, and so it doesn't carry out the operations until you execute `evalm`. When the `evalm` function is executed, it does the matrix multiplication properly. Thus, to compute $Av + b$, we would write `evalm(A &* v + b)`. Because we want to reduce everything modulo `p`, we `map modp` onto the matrix. Since we might want to specify an element as the inverse of another, we allow our matrix and vector to contain rationals. As before, we accept several types of inputs.

```
> with(linalg):
  AffineMatEncode :=
   proc(text::string,
        A::{matrix(rational),Matrix(rational),list(list(rational))},
        B::{vector(rational),Vector(rational),list(rational)})
   local   vtext,vcrypt,i,n,p;
   global  Alphabet;

   p:=length(Alphabet);
   n:=nops(convert(B,list));  # look at B to get the size

   if ( gcd(linalg[det](A),p)>1) then
       error("Determinant of %1 is not prime to the length of Alphabet %2",
             A,p);
   fi;

   vtext := StringToVects(text, n);
   vcrypt:= [seq( map(modp,
                      evalm(  A &* vtext[i]  + B),
                      p),
                 i=1..nops(vtext))];
   VectsToString(vcrypt);
  end:
```

Note that matrices allow the encodings of nearby characters in the message to interact. For example, we can exchange every other character in a message using the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$:

We should point out that all the examples in this section are using the 53-character alphabet of the previous section.

```
> AffineMatEncode("I am so mixed up", [[0,1],[1,0]], [0,0]);
```

“ Imas oimex dpu”

To decode a message encoded using A, b, we can encode the crypttext with the matrix A^{-1} and the shift vector $-A^{-1}b$. You should check for yourself why this is the proper transformation. To obtain the inverse matrix in Maple, you can use `inverse`. This is part of the linear algebra package, so you must either first issue the command `with(linalg,inverse)` or refer to the function as `linalg[inverse]`.

```
> A := matrix([[12, 27], [46, 44]]); B:=[25, 50];
```

$$A := \begin{bmatrix} 12 & 27 \\ 46 & 44 \end{bmatrix}$$

$$B := [25, 50]$$

```
> AffineMatEncode("Lester Hill",A, B);
```

xMPKFUGRxtaa

```
> AffineMatEncode(%, inverse(A), evalm(-inverse(A) &* B));
```

“Lester Hill ”

8.3 A Known-plaintext attack on an affine matrix cipher

As in §7.3, we can determine the key if we know what certain characters correspond to. If the message is encoded using n-vectors, we will need to know the decodings for $n+1$ different vectors, that is, $n(n+1)$ characters. We then solve the $n+1$ vector equations simultaneously to find out the deciphering matrix and shift vector.

We will give a small example: Suppose we receive a message that we know was encoded using an affine cipher on 2-vectors, in our usual 97-letter alphabet. Suppose we also know that the plaintext “secret” corresponds to the ciphertext “cilbup”. This is sufficient information to decipher the message, because we know the encodings of three different 2-vectors.

```
> Alphabet := cat("\n\t", Select(IsPrintable,convert([seq(i,i=1..255)],bytes))):
  plain:=StringToVects("secret",2);
```

$$plain := [[85, 71], [69, 84], [71, 86]]$$

```
> crypt:=StringToVects("cilbup",2);
```

$$crypt := [[69, 75], [78, 68], [87, 82]]$$

These three vectors give us three equations to solve:

$$Ac_1 + B \equiv p_1 \qquad Ac_2 + B \equiv p_2 \qquad Ac_3 + B \equiv p_3$$

where the c_i are the crypttext vectors and the p_i are the corresponding plaintext, and the equations are taken mod 97. (If this seems backwards, remember we are trying to find the deciphering transformation.) If the code were linear, then the deciphering matrix would be PC^{-1}, where P and C are the matrices whose columns are made up of the vectors p_i and c_i respectively.[26]

Since the code is affine, we can either play around manipulating matrices, or we can just translate everything into 6 linear equations in 6 variables. We will take the latter approach.

```
> A:=matrix([[a,b],[c,d]]);
  B:=[e,f];
```

$$A := \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$B := [e, f]$$

Note that we wish to solve $Ac_i + B \equiv p_i \pmod{97}$ for A and B. If we execute the statement `evalm(A &* crypt[i] + B - plain[i])`, we will get a vector which should be the zero vector. For example, using the first sample, we obtain

```
> evalm( A&*crypt[1] + B - plain[1]);
```

$$[69\,a + 75\,b - 85 + e,\ 69\,c + 75\,d - 71 + f]$$

This gives us a vector of two expressions we would like to set to zero. If we do this for all three of our samples, and unwrap the vectors using `op` and `convert( ,list)`, we will get a system of equations appropriate to solve:

```
> eqns:= {seq(op(convert(
                    evalm(A&*crypt[i] + B - plain[i]),
                    list)), i=1..3)};
```

$$eqns := \{87\,c + 82\,d - 86 + f,\ 87\,a + 82\,b - 71 + e,\ 78\,c + 68\,d - 84 + f,$$
$$78\,a + 68\,b - 69 + e,\ 69\,c + 75\,d - 71 + f,\ 69\,a + 75\,b - 85 + e\}$$

Now, we use `msolve` to find out the decoding matrix and shift vector.

[26]Do you see why this is so? Hint: think about the relationship between matrices and linear functions.

```
> Answer:= subs(msolve(eqns,97),[evalm(A),B]);
```

$$Answer := [\begin{bmatrix} 42 & 84 \\ 19 & 78 \end{bmatrix}, [5, 88]]$$

We can decode our message now, by encoding the crypttext using the matrix A and shift B found above. A is the inverse of the matrix used for encoding, and B is A times the negative of the original shift vector.

```
> AffineMatEncode("cilbup",Answer[1],Answer[2]);
```

"secret"

It is worth noting that combining an affine matrix cipher with a large multibyte alphabet (see §5.3) greatly increases its security, at least if one restricts to short messages, each using different keys. While the matrix cipher is still susceptible to a known-plaintext attack, getting the plaintext becomes much harder.

9 Modern cryptography

9.1 Secure cryptosystems

As mentioned before, most of the ciphers we have examined are not really cryptographically secure, although they may have been adequate prior to the widespread availability of computers. A cryptosystem is called secure if a good cryptanalyst, armed with knowledge the details of the cipher (but not the key used), would require a prohibitively large amount of computation to decipher a plaintext. This idea (that the potential cryptanalyst knows everything but the key) is called Kerckhoff's Law or sometimes Shannon's Maxim.

Kerckhoff's Law at first seems unreasonably strong rule; you may be thinking that, given a mass of encrypted data, how is the cryptanalyst to know *by what means* it was encrypted? If you are encrypting your own files for personal use and writing your own software which you keep under lock and key, this assumption is not unreasonable. But today, most encryption is done by software or hardware that the user did not produce himself. One can reverse engineer a piece of software (or hardware) and make the cryptographic algorithm apparent, so we cannot rely on the secrecy of the method alone as a good measure of security. For example, when you make a purchase over the internet, the encryption method that is used between your browser and the seller's web server is public knowledge. Indeed, it must be public: if not, only those privy to the secret could create web browsers or web servers. The security of the transaction depends on the security of the keys.

There are several widely used ciphers which are believed to be fairly secure. We will not go into the details of any of these; most of the algorithms are quite involved and require manipulations at the bit level— Maple is not really the appropriate tool for them. Implementations are widely available, both commercially and as free public software (see, for example, the International Cryptography page[27]).

Probably the most commonly used cipher today is DES (the Data Encryption Standard), which was developed in the 1970s and has been adopted as a standard by the US government. The standard implementation of DES operates on 64-bit blocks (that is, it uses an alphabet of length 2^{64}— each "character" is 8 bytes long), and uses a 56-bit key. Unfortunately, the 56-bit key means that DES, while secure enough to thwart the casual cryptanalyst, is attackable with special hardware by governments, major corporations, and probably well-heeled criminal organizations. One merely needs to try a large number of the possible 2^{56} keys.[28] This is a formidable, but not insurmountable, computational effort. A common variation of DES, called Triple-DES, uses three rounds of regular DES with three different keys (so the key length is effectively 168 bits), and is considerably more secure.

DES was originally expected to be used for "only a few years" when it was first designed. However, due to its surprising resistance to attack it was generally unassailable for nearly 25 years. The powerful methods of linear and differential cryptanalysis were developed to attack block ciphers like DES.

In the January of 1997, the National Institute for Standards and Technology (NIST) issued a call for a new encryption standard to be developed, called AES (the Advanced Encryption Standard). The requirements were that they operate on 128-bit blocks and support key sizes of 128, 192, and 256 bits. There were five ciphers which advanced to the second round: MARS, RC6, Rijndael, Serpent, and Twofish. All five were stated to have "adequate security"– Rijndael was adopted as the standard in October of 2000. Rijndael was created by the Belgian cryptographers Joan Daemen and Vincent Rijmen; the underlying mathematics is the algebra of polynomials in the finite field $GF(2)$.

Other commonly used ciphers are IDEA (the International Data Encryption Algorithm, developed at the ETH Zurich in Switzerland), which uses 128-bit keys, and Blowfish (developed by Bruce Schneier), which uses variable length keys of up to 448 bits. Both of these are currently believed to be secure. Another common cipher, RC4 (developed by RSA Data Security) can use variable length keys and, with sufficiently long keys, is believed to be secure. Some versions the Netscape web browser used RC4 with 40 bit keys for secure communications. A single encrypted session was broken in early 1995 in about 8 days using 112 networked computers (see Damien Doligez's web page about cracking SSL[29] for more details); later in the same year a second session was broken in under 32 hours. Given the speed increases in computing since then, it is reasonable to believe that a 40-bit key can be cracked in a few hours. Notice, however, that both of these attacks were essentially brute-force, trying a large fraction of the 2^{40} possible keys. Increasing the key size resolves that problem. Nearly all browsers these days use at least 128-bit keys.

[27]http://www.cs.hut.fi/ssh/crypto/
[28]that is, 72,057,594,037,927,936 keys.
[29]http://pauillac.inria.fr/ doligez/ssl/

There are, of course, many other ciphers in common use, and more being developed all the time.

9.2 Message digests

Related to encryption are message digests, or cryptographic hash functions. A message digest takes an arbitrary length string and computes a number, or hash, from it. Changing the string changes the value of the hash. Hashing functions also play a major role in searching and sorting, and hence in computer databases. While it is obvious that several plaintexts must correspond to the same hash value, it is a difficult problem in general to construct such a plaintext from the value of the hash alone. Thus, one can use a good message digest to authenticate a message—appending an encrypted version of the hash to an otherwise clear text message can act as a seal, showing that the message was not otherwise tampered with, since it would be infeasably hard to construct a different message with the same hash.

Some commonly used Message digests are MD5 (developed by RSA Data Security), and SHA or SHS (the Secure Hash Algorithm/Standard, developed by the US government).

It is worth remarking here that computer systems can use a cryptographic hash function as a way to store a password (for example, Linux allows the use of MD5 for passwords). A computer password never need be decrypted; it only needs to be verified. Instead of storing the password itself, or an encrypted version of it, the computer stores the corresponding hash. Then, when a user signs in, the password is hashed and compared to the stored value. If they match, the user typed the right password (or managed to find another password with the same hash, which is computationally impossible).

9.3 Public Key cryptography

All of the ciphers so far discussed are symmetric encryption routines (also called secret-key or traditional ciphers): if one knows the key used to encode the message, one also can decrypt the message without too much work. For example, in an affine cipher $x \mapsto ax + b \bmod p$ (see §7), if we know the values of a and b used to encode the message, we can quickly compute the deciphering key $1/a \bmod p$, $-b/a \bmod p$. By the same token, the ability to decrypt the message usually also yields the key used for encryption.[30]

[30] A story related to this appears in Casanova's memoirs (1757) (quoted in [Kahn]). He was given an encrypted manuscript by a Madame d'Urfé, which he deciphered. From deciphering, he determined the key used to encipher the manuscript. Later, upon learning this, Madame d'Urfé was incredulous, believing he must have learned the keyword in order to decode the message.

> I then told her the key-word, which belonged to no language, and I saw her surprise. She told me that it was impossible, for she believed herself the only possessor of that word which she kept in her memory and which she had never written down.
>
> I could have told her the truth— that the same calculation which had served me for deciphering the manuscript had enabled me to learn the word— but on a caprice it struck me to tell her that

This was true of all cryptosystems up until the mid 1970s: knowledge of how to encode a message allowed one also to decipher it. However, in 1976 Whitfield Diffie and Martin Hellman invented public key cryptography [DH76]. In a public key system, someone who knows how to encipher a message cannot determine how to decipher the message without a prohibitively large computation. That is, given the enciphering transformation $f_{key} : \mathcal{P} \mapsto \mathcal{C}$, discovering f_{key}^{-1} is not computationally realistic. It is important to realize that there may be a procedure for doing so, but to carry out this process would take a prohibitively long time (say, hundreds of years on the fastest known computers). Such a function is sometimes called a one-way function. However, to be usable for public key cryptography, it must be possible to find the inverse of the function provided you have some extra information. This extra information is referred to as a trapdoor (like a trapdoor that a magician would use to escape from a "sealed" box), resulting in a trapdoor one-way function.

What good is a public key cryptosystem? Since knowledge of the encoding key K_E does not give information about the decoding key K_D, the encoding key K_E can be made public (hence the name public key). This allows anyone to encode messages that only the recipient can decode. For example, suppose you want to make a credit card purchase over the Internet. Data sent across the Internet unencrypted is not secure: someone with access to the transmission lines can listen in and capture sensitive data.[31] However, if the merchant provides a public key, you can encrypt your credit card number and transmit it without worry. Only the merchant can decrypt the message, even though anyone may send one.[32]

Another feature of public key cryptography is authentication. In many public key systems, either the enciphering key K_E or the deciphering key K_D may be made public without revealing the other one. In order to digitally sign a message, I could append my name encrypted with my enciphering key, which I keep secret. I could also make public the deciphering key K_D. Then anyone can check that I was the actual sender, because only I could have encoded my name. Of course, to stop someone from merely appending a valid encrypted signature to a forged message, I should also encrypt something specific to that message, such as a message digest (see §9.2) computed on its contents. This feature is commonly used to digitally sign email by

> a genie had revealed it to me. This false disclosure fettered Madame d'Urfé to me. That day, I became the master of her soul, and I abused my power. Every time I think of it, I am distressed and ashamed, and I do penance now in the obligation under which I place myself of telling the truth in writing my memoirs.

[31]This is not as hard as you might expect. In many places, such as universities, portions of the network backbone pass through public places like dormitories or classrooms. One merely needs to attach a computer to the network lines to "sniff" the network traffic. This will, of course, be limited to users communicating to or from that location, but that can be quite a lot. This same problem occurs with some cable-modem providers: appropriate software can detect the network traffic of one's neighbors. Also, there are many segments of the Internet which consist of microwave links. To listen in on a huge part of network traffic, all you need is an appropriately placed receiver.

[32]Of course, you should still only deal with reputable merchants. You probably wouldn't give your credit card number to a guy selling stuff on the street; neither should you automatically trust anyone with a web site.

software such as PGP[33].

Public-key cryptography is very computationally intensive, so typically its use is limited to allow for the secure transmission of a secret key; this secret key is then used to encrypt the rest of the message using a symmetric encryption method such as AES, triple-DES or IDEA.

A common public key system in current use is RSA (named for its inventors: Ronald Rivest, Adi Shamir and Leonard Adelman), which we will look at in detail in §11. Others are the Diffie-Hellman key exchange system, El Gamal, Subset-sum (or Knapsack) systems, and systems based on elliptic curves. We will not discuss these here.

10 Some Number Theory

Most public key systems rely on number-theoretic results. Before we can discuss the implementation of one, we need to quickly go over the necessary background. We have already used a tiny amount of number theory (in our discussion of computing $\bmod p$ and of the greatest common divisor). Of course, this must be done briefly, and we will only touch on a small part of a large and ancient field— the interested reader would do well to consult a text on number theory (e.g. [NZM], [Ros]) for more information.

10.1 The greatest common divisor and the Euclidean algorithm

We have already met the greatest common divisor, or gcd, which is the largest integer which divides both of a pair of numbers. Two numbers are said to be relatively prime if their greatest common divisor is 1. As we have already seen, finding two relatively prime numbers has important applications in many cryptosystems.

How can we determine the gcd of two numbers? If the numbers are not too large, just looking at their factors does the trick. For example,

$$\gcd(138, 126) = 6 \quad \text{since} \quad 138 = 2 \cdot 3 \cdot 23 \quad \text{and} \quad 126 = 2 \cdot 3^2 \cdot 7.$$

However, there must be a more efficient way, since Maple can calculate the gcd of two 60-digit numbers in well under a second, while taking a long time (about 10 minutes on a 600 Mhz pentium computer) to factor either one of them.

One such algorithm is the Euclidean algorithm, which has been around for thousands of years. It works by computing successive differences— to compute $\gcd(a, b)$, where $a > b$, we first compute $r_1 = a - k_1 b$, where $k_1 b$ is the largest multiple of b that is less than a. Then we repeat this, finding $r_2 = b - k_2 r_1$, $r_3 = r_1 - k_3 r_2$, and so on until the remainder is zero. The last nonzero remainder is the gcd. For example, to calculate $\gcd(138, 126)$,

[33] http://www.pgpi.com

$$\begin{array}{rcrrcrcl} & & a & & = & 138 & & \\ & & & b & = & 126 & & \\ & & a & -b & = & 12 & = & r_1 \\ b - 10r_1 & = & -10a & +11b & = & 6 & = & r_2 \\ r_1 - 2r_2 & = & 21a & -23b & = & 0 & & \end{array}$$

This not only gives $\gcd(138, 126) = 6$, but expresses it as a difference of multiples of the two numbers: $6 = 11 \times 126 - 10 \times 138$. We can write this procedure more formally as a theorem.

Theorem 10.1 (The Euclidean Algorithm) *Let a and b be two positive integers, with $b < a$. Then we can construct two decreasing sequences of positive integers r_i and k_i as follows:*

$$\begin{array}{lclcll} a & - & k_1 b & = & r_1 & 0 < r_1 < b \\ b & - & k_2 r_1 & = & r_2 & 0 < r_2 < r_1 \\ r_1 & - & k_3 r_2 & = & r_3 & 0 < r_3 < r_2 \\ r_2 & - & k_4 r_3 & = & r_4 & 0 < r_4 < r_3 \\ & & & \vdots & & \\ r_{n-2} & - & k_n r_{n-1} & = & r_n & 0 < r_n < r_{n-1} \\ r_{n-1} & - & k_{n+1} r_n & = & 0 & \end{array}$$

and r_n is the greatest common divisor of a and b.

We have already seen that the Euclidean algorithm also gives us the following:

Corollary 10.2 *Let a and b be two positive integers. Then there are integers x and y so that $ax + by = \gcd(a, b)$.*

Why does the Euclidean Algorithm work? It follows from the observation that if $a = bk + r$, then $\gcd(a, b) = \gcd(b, r)$. For example, $\gcd(138, 126) = \gcd(126, 12) = \gcd(12, 6) = 6$. We write this as a lemma.

Lemma 10.3 *Let b be a positive integer, and a be a nonnegative integer. If*

$$a = bk + r, \qquad \text{with } k \text{ and } r \text{ positive integers}$$

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

Proof. Let $d = \gcd(a, b)$. Then since d is a divisor of a and a divisor of b, it is also a divisor of $a - bk$. Since $r = a - bk$, we have d a divisor of r. So it must divide $\gcd(b, r)$.

Now, in order to prove equality, we want to show that also $\gcd(b, r)$ is a divisor of d. Certainly $\gcd(b, r)$ divides a, since $a = r + bk$. It also divides b, and so we have $\gcd(b, r)$ as a divisor of $\gcd(a, b)$.

Thus, $\gcd(b, r) = \gcd(a, b)$. □

Now we are ready to prove that the Euclidean Algorithm (Thm. 10.1) always yields the greatest common divisor.

Proof. Notice that the sequence $b > r_1 > r_2 > r_3 > \ldots$ is a strictly decreasing sequence of positive integers, so it must eventually terminate with some $r_n > 0$. The last equation can be written as $k_{n+1}r_n = r_{n-1}$, and so r_n is a divisor of r_{n-1}. Then certainly $r_n = \gcd(r_n, r_{n-1})$. Applying the lemma to the equation above it, we obtain $\gcd(r_n, r_{n-1}) = \gcd(r_{n-1}, r_{n-2})$. Repeating in this way, we obtain $r_n = \gcd(r_n, r_{n-1}) = \gcd(r_{n-1}, r_{n-2}) = \ldots = \gcd(r_1, b) = \gcd(a, b)$. □

Recall that if r is such that $ar \equiv 1 \pmod{n}$, then r is called the multiplicative inverse of a modulo n. We have already dealt with such objects extensively. We now state a theorem which tells us exactly when such inverses exist. Note that this is really just Prop. 7.1 slightly reworded.

Theorem 10.4 *Let a and n be integers, with $n \geq 2$. Then a has a multiplicative inverse modulo n if and only if* $\gcd(a, n) = 1$.

Proof. First, suppose that a has an inverse modulo n. Then there is a k so that

$$ak \equiv 1 \pmod{n}.$$

In particular, n is a divisor of $ak - 1$, so there must be some integer t so that $ak - 1 = nt$. Rewriting this as

$$ak - nt = 1,$$

we see that $\gcd(a, n) = 1$ (by Cor. 10.2).

Now suppose that $\gcd(a, n) = 1$. Then again using Cor. 10.2, there are integers r and s with $ar + ns = 1$. This means $ar - 1$ is divisible by n, so

$$ar \equiv 1 \pmod{n}.$$

But this says that r is the inverse of a modulo n. □

Notation. It is common (and quite convenient) to denote the set of integers modulo n as $\mathbb{Z}_n$. That is,

$$\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$$

As we already know, this set is closed under addition and multiplication, and the usual associative, distributive, and commutativity laws hold. This set is an example of an algebraic structure called a ring.

Notation. We will use $\mathbb{Z}_n^*$ to denote the elements of $\mathbb{Z}_n$ which have multiplicative inverses. In light of the last theorem, we have

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid \gcd(a, n) = 1\}$$

For example,

$$\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\} \qquad \mathbb{Z}_8^* = \{1, 3, 5, 7\} \qquad \mathbb{Z}_{12}^* = \{1, 5, 7, 11\}$$

10.2 The Chinese Remainder Theorem

In this section, we state a very old theorem about simultaneous solutions to congruences. This theorem may have been known to the eight-century Buddhist monk I-Hsing, and certainly appears in Ch'in Chiu-shao's *Mathematical Treatise in Nine Sections*, written in 1247. It is sometimes (rarely) referred to as the "Formosa Theorem".

Theorem 10.5 *Let $m \geq 2$ and $n \geq 2$ be integers with* $\gcd(m, n) = 1$*. Then for any integers a and b, the system of equations*

$$\begin{aligned} x &\equiv a \mod m \\ x &\equiv b \mod n \end{aligned}$$

has a solution. Furthermore, there is only one solution x between 0 and $mn - 1$.

As an example of this, suppose five kids pool all their money and buy a bag of candy. They divide the candy up evenly, and find there are 3 pieces left over. Suddenly, one kid's mom comes out, tells him "no candy before dinner", and hauls him off. The other four kids divide up the candy and there is one piece left over. While they are squabbling over it, a dog runs up and eats the disputed piece.

The Chinese Remainder Theorem tells us that we can always determine how many pieces of candy were in the bag, at least up to multiples of 20. In the above example, there were 13 pieces of candy (or 33, or 53, or ...). If we want to solve this problem with Maple, the command `chrem([3,1],[5,4])` does the trick. We could also use `numtheory[mcombine](5,3,4,1)`.

Proof. Since m and n are relatively prime, there are integers k and t so that $mk + nt = 1$. Then

$$c = bkm + ant$$

is the desired solution. To verify this, we need only check:

$$c \equiv ant \equiv a(1 - mk) \equiv a \pmod{m}$$

and

$$c \equiv bkm \equiv b(1 - nt) \equiv b \pmod{n}$$

To check uniqueness, suppose there are two solutions c and d. Since $c \equiv a \pmod m$ and $d \equiv a \pmod m$, we have $c - d \equiv 0 \pmod m$. Similarly, $c - d \equiv 0 \pmod n$. Since both m and n divide $c - d$, and m and n are relatively prime, we have that mn divides $c - d$. Thus $c - d \equiv 0 \pmod{mn}$, that is, $c \equiv d \pmod{mn}$. □

10.3 Powers modulo n

Notation. In this and future sections, we shall use the notation $[\![a]\!]_b$ to denote $a \bmod b$.

First, let's take a look at a few examples of what happens when we reduce the sequence $a, a^2, a^3, a^4, \ldots$ modulo n. Consider $[\![3^k]\!]_{20}$:

$$3,\ [\![3^2]\!]_{20} = 9,\ [\![3^3]\!]_{20} = 7,\ [\![3^4]\!]_{20} = [\![21]\!]_{20} = 1,\ [\![3^5]\!]_{20} = 3, \ldots$$

The sequence consists of four numbers $3, 9, 7, 1$ and then it repeats. What about 4? We have

$$4,\ [\![4^2]\!]_{20} = 16,\ [\![4^3]\!]_{20} = 4,\ [\![4^4]\!]_{20} = 16, \ldots$$

This is also periodic, but notice that we never get back to 1. Similarly, let's look at the powers of 6:

$$6,\ [\![6^2]\!]_{20} = [\![36]\!]_{20} = 16,\ [\![6^3]\!]_{20} = [\![16 \cdot 6]\!]_{20} = [\![96]\!]_{20} = 16,\ [\![6^4]\!]_{20} = 16, \ldots$$

This gets stuck at 16. Finally, consider the powers of 11:

$$11,\ [\![11^2]\!]_{20} = [\![121]\!]_{20} = 1,\ [\![11^3]\!]_{20} = 11, \ldots$$

This sequence is periodic of period 2, and contains 1.

Definition 10.6 *If there is a positive integer k so that $[\![a^k]\!]_n = 1$, we say that a has* finite multiplicative order *modulo n. The smallest such integer k is called the* order of *a modulo n.*

We can readily categorize the elements with finite order. If you look at the examples above, 3 and 11 have finite order mod 20, while 4 and 6 do not. You probably have already guessed that this is because $\gcd(3, 20) = 1$ and $\gcd(11, 20) = 1$, while $\gcd(4, 20) = 4$ and $\gcd(6, 20) = 2$.

Theorem 10.7 *Let a and $n > 2$ be integers. Then a has finite order modulo n if and only if* $\gcd(a, n) = 1$.

Proof. First, suppose a has finite order mod n. Then there is some k so that

$$1 = [\![a^k]\!]_n = [\![a]\!]_n \cdot [\![a^{k-1}]\!]_n.$$

Thus, a has a multiplicative inverse, namely $\llbracket a^{k-1} \rrbracket_n$. By Thm. 10.4, this means $\gcd(a, n) = 1$.

Now we suppose $\gcd(a, n) = 1$, and we want to find the inverse of a. Consider the sequence

$$a, \llbracket a^2 \rrbracket_n, \llbracket a^3 \rrbracket_n, \ldots, \llbracket a^n \rrbracket_n, \llbracket a^{n+1} \rrbracket_n.$$

Since there are $n+1$ elements and we are reducing modulo n, there must be at least two which are equal. Let's denote these by $\llbracket a^k \rrbracket_n$ and $\llbracket a^t \rrbracket_n$, with $1 \le t < k$. Since

$$\llbracket a^k \rrbracket_n = \llbracket a^t \rrbracket_n$$

we have

$$\llbracket a^{k-t} \rrbracket_n = 1.$$

Thus, a has order $k - t$ mod n. □

So, we have seen that any invertible element of $\mathbb{Z}_n$ has finite order. In particular, if p is a prime number, every element except 0 is invertible and so has finite order. What are the possible orders? A theorem of Pierre de Fermat gives a result in that direction.

Theorem 10.8 (Fermat's Little Theorem, 1640) *Let p be a prime. If $\gcd(a, p) = 1$, then*

$$a^{p-1} \equiv 1 \ (mod p).$$

For any integer a,

$$a^p \equiv a \ (mod p).$$

Among other things, this theorem says that if p is prime, then the order of a mod p always divides $p-1$. Note that it doesn't say the order *is* $p-1$, just that it is a divisor of it. Also, it says nothing about order with modulo a non-prime. We won't give the proof of this theorem here. Rather, we will give a proof of a more general result in the next section.

10.4 The Euler φ-function and Euler's Theorem

Recall that we denote the set of invertible elements of $\mathbb{Z}_n$ by $\mathbb{Z}_n^*$. That is,

$$\mathbb{Z}_n^* = \{a \in Z_n \mid \gcd(a, n) = 1\}.$$

This set is sometimes called a reduced residue system modulo n.

Definition 10.9 *Let n be a positive integer. Then $\varphi(n)$ is the number of elements in $\mathbb{Z}_n^*$, that is, the number of positive integers less than n which are relatively prime to n. This function is called the* Euler φ-function,[34] *or sometimes Euler's totient function.*

[34]Although the function was invented by Euler, the notation $\varphi(n)$ to represent it is actually due to Gauss.

We can compute a few values of $\varphi(n)$ just by counting.

$$\begin{array}{rcrlcll}
\varphi(2) &=& 1 & \text{since} & \mathbb{Z}_2^* &=& \{1\} \\
\varphi(3) &=& 2 & \text{since} & \mathbb{Z}_3^* &=& \{1,2\} \\
\varphi(4) &=& 2 & \text{since} & \mathbb{Z}_4^* &=& \{1,3\} \\
\varphi(5) &=& 4 & \text{since} & \mathbb{Z}_5^* &=& \{1,2,3,4\} \\
\varphi(6) &=& 2 & \text{since} & \mathbb{Z}_6^* &=& \{1,5\} \\
\varphi(7) &=& 6 & \text{since} & \mathbb{Z}_7^* &=& \{1,2,3,4,5,6\} \\
\varphi(8) &=& 4 & \text{since} & \mathbb{Z}_8^* &=& \{1,3,5,7\} \\
\varphi(9) &=& 6 & \text{since} & \mathbb{Z}_9^* &=& \{1,2,4,5,7,8\} \\
\varphi(10) &=& 4 & \text{since} & \mathbb{Z}_{10}^* &=& \{1,3,7,9\} \\
\varphi(11) &=& 10 & \text{since} & \mathbb{Z}_{11}^* &=& \{1,2,3,4,5,6,7,8,9,10\} \\
\varphi(12) &=& 4 & \text{since} & \mathbb{Z}_{12}^* &=& \{1,5,7,11\}
\end{array}$$

Of course, Maple also knows about the function φ in the `numtheory` package, so we could generate more values by asking Maple. But let's try to understand a bit more. We can readily prove some properties of φ.

Proposition 10.10 *Let $p \geq 2$ be a prime. Then*

a. $\varphi(p) = p - 1$.

b. *If n is a positive integer, then $\varphi(p^n) = p^n - p^{n-1}$.*

c. *If a and b are relatively prime, then $\varphi(ab) = \varphi(a)\varphi(b)$.*

Proof. (a) This is immediate, since all non-zero elements of $\{0, 1, \ldots, p-1\}$ are invertible.

(b) We prove this part just by counting. Since p is prime, the only numbers in $\mathbb{Z}_p$ which are not relatively prime to p^n are multiples of p. These are $0, p, 2p, 3p, \ldots, p^2, (p+1)p, \ldots, (p^{n-1}-1)p$. Note that $p^{n-1}p = p^n$, so there are exactly p^{n-1} multiples of p less than p^n (including 0). All the others are relatively prime to p— this gives the result.

(c) We need to show that the number of elements in $\mathbb{Z}_{ab}^*$ is the same as the product of the size of $\mathbb{Z}_a^*$ and $\mathbb{Z}_b^*$. We do that by showing that for each $t \in \mathbb{Z}_{ab}^*$, there is exactly one pair (r, s) with $r \in \mathbb{Z}_a^*$ and $s \in \mathbb{Z}_b^*$.

So, let $r \in \mathbb{Z}_a^*$ and $s \in \mathbb{Z}_b^*$. Then, by the Chinese Remainder Theorem (Thm. 10.5), there is an integer $t < ab$ with

$$t \equiv r \bmod a \qquad \text{and} \qquad t \equiv s \bmod b.$$

Since t is a solution of the first, we can find an integer k so that $r = t + ka$, so $\gcd(t, a) = \gcd(a, r) = 1$ by Lemma 10.3. Similarly, $\gcd(t, b) = 1$. Since t is relatively prime to both a and b, it is relatively prime to their product, so $\gcd(t, ab) = 1$. Thus $t \in \mathbb{Z}_{ab}^*$.

Now take $t \in \mathbb{Z}^*_{ab}$, and we must find a corresponding r and s. Let $r = [\![t]\!]_a$. Now, since $\gcd(t, ab) = 1$, certainly $\gcd(t, a) = 1$. Since $r \equiv t \bmod a$, there is an integer k so that $t = r + ka$, so we have $\gcd(r, a) = 1$. Thus $r \in \mathbb{Z}^*_a$. Similarly, we let $s = [\![t]\!]_b$, and we see that $s \in \mathbb{Z}^*_b$.

Since we have paired each $t \in \mathbb{Z}^*_{ab}$ with a unique pair $(r, s) \in \mathbb{Z}^*_a \times Z^*_b$, they have the same cardinality. The first set has $\varphi(ab)$ elements, and the latter has $\varphi(a)\varphi(b)$. □

We now come to Euler's generalization of Fermat's result. This result is the central idea underlying the RSA public key cryptosystem.

Theorem 10.11 (Euler, 1750) *Let a and n be relatively prime integers, with $n \geq 2$. Then*

$$[\![a^{\varphi(n)}]\!]_n = 1.$$

Proof. Let a be relatively prime to n, and consider the set

$$a\mathbb{Z}^*_n = \left\{ [\![ab]\!]_n \,\middle|\, b \in \mathbb{Z}^*_n \right\}.$$

Since a and n are relatively prime, $[\![a]\!]_n \in \mathbb{Z}^*_n$. Thus every element of $a\mathbb{Z}^*_n$ is in $\mathbb{Z}^*_n$. But also every element of $\mathbb{Z}^*_n$ is an element of $a\mathbb{Z}^*_n$, since $[\![ab]\!]_n = [\![ac]\!]_n$ if and only if $[\![b]\!]_n = [\![c]\!]_n$. This means that the sets $a\mathbb{Z}^*_n$ and $\mathbb{Z}^*_n$ are equal.

Now, multiply all the elements of $\mathbb{Z}^*_n$ together; call the result N. Then $[\![N]\!]_n \in \mathbb{Z}^*_n$. If we multiply all the elements of $a\mathbb{Z}^*_n$ together, we get $a^{\varphi(n)}N$, and since the two sets are the same, we have

$$[\![a^{\varphi(n)}N]\!]_n = [\![N]\!]_n.$$

Thus,

$$[\![a^{\varphi(n)}]\!]_n = 1.$$

□

11 The RSA Public key cryptosystem

In 1978, R. Rivest, A. Shamir, and L. Adelman published the description of the RSA public key system[RSA], which we will describe here. As discussed in §9.3, a public key system allows anyone to encode messages that only the intended recipient may decode. This algorithm relies on the fact that unless one knows some special facts about n, calculating the Euler φ-function $\varphi(n)$ is as hard as factoring n, which, for very large n (say, 200 digits), is very hard indeed.

11.1 Details of the RSA algorithm

To set up the system, we pick at random two large primes p and q, of about 100 digits each. We then compute the base n as the product of p and q. Note that $\varphi(n) = (p-1)(q-1)$. We also pick some other number e, called the exponent, which is relatively prime to $\varphi(n)$. We make the numbers (n, e) public— these form the key needed for encoding. We also use the Euclidean algorithm to compute x which satisfies $ex - y\varphi(n) = 1$; that is, so that $[\![ex]\!]_{\varphi(n)} = 1$. The number x is the part of the key we keep private; our decrypting key is the pair (n, x).

To encode a message: The sender divides his message up into blocks of length k, and to each block assigns an integer β (for example, by treating the characters of the message as a k-graph, as in §5.3, with $0 \le \beta < n$. For each block of plaintext, the sender transmits $[\![\beta^e]\!]_n = \mu$.

To decode the message: For each unit μ of the crypttext received, the recipient computes $[\![\mu^x]\!]_n$, which is the integer representing of a k-graph of the original plaintext message. We can see that this is so, because

$$[\![\mu^x]\!]_n = [\![\beta^{ex}]\!]_n = [\![\beta^{1+\varphi(n)y}]\!]_n = [\![\beta]\!]_n [\![\beta^{y\varphi(n)}]\!]_n = [\![\beta]\!]_n \left([\![\beta^{\varphi(n)}]\!]_n\right)^y = [\![\beta]\!]_n \cdot 1^y = \beta.$$

This relies on Euler's theorem (Thm. 10.11), so that $[\![\beta^{\varphi(n)}]\!]_n = 1$. Euler's theorem only guarantees that this will work for $\gcd(\beta, n) = 1$. However, we can easily see that decryption works for all $\beta < n$, whether or not it is relatively prime to n.

Proposition 11.1 *Let $n = pq$, with p and q prime, and let $0 \le \beta < n$. Then for any integer y, $[\![\beta^{1+y\varphi(n)}]\!]_n = \beta$.*

Proof. If $\gcd(\beta, n) = 1$, the result follows from Euler's theorem.

Otherwise, since p and q are prime, we have β a multiple of either p or q, but not both (since $\beta < pq$ by assumption). Suppose $\beta \equiv 0 \pmod{p}$, and so

$$[\![\beta^{1+y\varphi(n)}]\!]_p = 0 = [\![\beta]\!]_p.$$

Since β is a multiple of p less than pq, and both p and q are prime, we must have $\gcd(\beta, q) = 1$. We can apply Euler's theorem using modulus q to conclude $[\![\beta^{\varphi(q)}]\!]_q = 1$, so

$$[\![\beta^{1+y\varphi(n)}]\!]_q = [\![\beta]\!]_q \left([\![\beta^{\varphi(q)}]\!]_q\right)^{y(p-1)} = [\![\beta]\!]_q.$$

We have two equations modulo p and modulo q, and can apply the Chinese Remainder Theorem (Thm. 10.5) to obtain one modulo pq. Hence

$$[\![\beta^{1+y\varphi(n)}]\!]_n = [\![\beta]\!]_n = \beta$$

as desired. In the case where $[\![\beta]\!]_q = 0$, the argument is the same after exchanging p and q. □

Remarks.

1. The symmetry between e and x means that we can encrypt with either the public or the private key, using the other for decryption. To use RSA works for authentication, the private key (n, x) is used encrypt a message which can be decrypted with the public key (n, e). Only one who possesses the private key could have encrypted such a message, which serves as a proof of identity.

2. If e is small, there is a possibility of having some message elements β for which $\beta^e < n$; for such a β, decrypting without knowledge of $\varphi(n)$ is easy: just compute the e-th root of β, since n is not a consideration. With care, one can either ensure that such β do not occur, or one can encrypt such messages in a slightly different manner.

3. For efficiency reasons, many implementations always use 3 for the encoding exponent e. In general, computing $[\![\beta^e]\!]_n$ when e is of the form $2^k + 1$ is quite efficient. Other common choices for e are 17 and 65537 ($2^{16} + 1$), which are only slightly less efficient than 3.

4. Instead of using the Euler φ-function, one can use Carmichael's λ-function, which is the largest order of any element of $\mathbb{Z}_n^*$. In the case where n is a product of two primes p and q, we have $\lambda(n) = \operatorname{lcm}(p - 1, q - 1)$. Since $p - 1$ and $q - 1$ are always even, the resulting private exponent x will be smaller.

5. There are a number of considerations in the choice of primes p and q that we will mostly ignore. For example, p and q should not be too close together, and both $p - 1$ and $q - 1$ should both have at least one large prime factor.

11.2 Implementing RSA in Maple

Now that we have covered how the RSA system works in theory, it is fairly straightforward to implement it.

Implementing the basics of RSA

First, we write a Maple procedure which, given a size for n (our public base), will choose n as the product of two large primes p and q, as well as an exponent e which is relatively prime to $\varphi(n)$, and compute the decrypting exponent x.

For real security, n should be at least 200 decimal digits, meaning that the primes p and q will be about 100 digits each. Such large numbers are not easily factored (which is how we get our security), so how do we expect to be able to find 100-digit prime numbers? It turns out that there are a number of very efficient ways to test whether a number is prime without actually factoring it. Maple's `isprime` function uses such methods. So all we really need to do is generate a random number of around 100 digits, then use `nextprime` to get the next prime bigger than that number. We should use `randomize()` to ensure that we get different numbers each time (See the discussion of pseudo-random numbers in §5.2, page **4**:17).

In `RSAsetup` below, the approximate number of digits for n is given, and p is chosen as a prime with one less than half that many digits, while q has one more than half the number of digits of n. This means n must have at least 5 digits (or else p would have to be 2, 3, 5 or 7). After calculating n and $\varphi(n)$, a random number is chosen for the public exponent e. If this number is not relatively prime to $\varphi(n)$, another random choice is taken until such a number is found. Finally, x is computed as the inverse of e modulo $\varphi(n)$. The result of the function is the public key (n, e) and the private key (n, x).

```
> randomize():

  RSAsetup:=proc(numdigits::posint)
    local p,q,phi,e,n,  x, pm, qm;

    if (numdigits <= 4) then
       error("Too few digits for this to work");
    fi;

    pm:=floor((numdigits-1)/2);
    qm:=numdigits-pm;
    p:=nextprime(rand(10^(pm-1)..10^(pm))());
    q:=nextprime(rand(10^(qm-1)..10^(qm))());

    n:=p*q;
    phi:=(p-1)*(q-1);

    e:=rand(3..phi/2)();
    while(gcd(e,phi) <> 1) do
      e:=rand(3..phi/2)();
    od;
    x := modp(1/e, phi);
    return([[n,e],[n,x]]);
  end:
```

We should remark that in certain cases, both p and q might be chosen at the extreme ends of their ranges, giving n with one fewer or one more digit than requested. If this were important, it could be repaired with a slightly more complicated program.

Once the keys are chosen, writing the heart of the RSA encoding is quite easy. We just need a function that eats a number β and a key (n, e), and computes $[\![\beta^e]\!]_n$.

```
> RSAEncodeNum := proc(num::nonnegint, key::list(posint))
    return( modp( num &^ key[2], key[1] ));
  end:
```

As a brief example, we'll generate a random pair of keys with a 5-digit modulus, and then encrypt and decrypt the number 47.

```
> keys:=RSAsetup(5):
  public := keys[1];
  private:= keys[2];
```

$$public := [10793, 2371]$$

$$private := [10793, 31]$$

```
> RSAEncodeNum(47,public);
  RSAEncodeNum(%,private);
```

$$3991$$

$$47$$

Of course, we want to encrypt more than a single number, but this is the heart of the whole thing.

Making it Useful

In order to encrypt a string of text, we have to convert our text to a list of numbers. It is probably best to use k-graphs to represent our text, as in §5.3, where k is the largest integer power of the size of our alphabet that is smaller than n. While it will work to use a single character alphabet, this can compromise security significantly— see the remarks at the end of §11.1.

We can use `StringToKgraph` and `KgraphToString` from §5.3 to represent our strings as k-graphs. In order to determine the appropriate choice of k, we find the largest integer k so that

$$\texttt{Alen}^k < n$$

where `Alen` is the length of our alphabet.

Once we have our plaintext represented as a list of integers, we merely `map` the function `RSAEncodeNum` onto that list.

```
> Alphabet := cat("\n\t",Select(IsPrintable,convert([seq(i,i=1..255)],bytes))):
  Alen := length(Alphabet);
  k:=floor(log[Alen](public[1]));
```

$$Alen := 97$$

$$k := 2$$

```
> text := "Once upon a midnight dreary, while I pondered, weak and weary":
```

```
> StringToKgraph(text,k);
```

[7809, 6956, 8441, 7939, 274, 261, 7354, 7830, 7156, 8416, 6792, 6971, 8215, 1449, 8635, 7349, 6965, 4173, 7956, 7841, 6957, 6971, 1428, 8635, 6570, 271, 7827, 264, 6976, 8215, 91]

```
> crypt:=map(RSAEncodeNum,%,public);
```

crypt := [23972, 14519, 16728, 21226, 15537, 16168, 23547, 21311, 657, 15151, 20048, 7796, 961, 8740, 4392, 5615, 18965, 18341, 18628, 14070, 9286, 7796, 23471, 4392, 17418, 12708, 3247, 11788, 23561, 961, 9606]

We can reverse the process, using the private key, to decrypt.

```
> KgraphToString(
       map(RSAEncodeNum,crypt,private),
       k);
```

"Once upon a midnight dreary, while I pondered, weak and weary"

It is important to note that if we want to represent the result of the encryption as a string instead of a list of numbers, we must use $(k+1)$-graphs rather than k-graphs. This is because $\texttt{Alen}^k < n$, so there will often be β so that $[\![\beta^e]\!]_n > \texttt{Alen}^k$. In the example above, note that $97^2 = 9409$, and there are several elements of the list `crypt` larger than 9409.

We can put it all together in two simple procedures.

```
> RSAEncodeString := proc(text::string, key::list(posint))
     local Alen, k;
     global Alphabet;

     Alen := length(Alphabet);
     k:=floor(log[Alen](key[1]));
     KgraphToString(
         map(RSAEncodeNum,
             StringToKgraph(text, k),
             key),
         k+1);
  end:

  RSADecodeString := proc(text::string, key::list(posint))
     local Alen, k;
     global Alphabet;

     Alen := length(Alphabet);
     k:=floor(log[Alen](key[1]));
     KgraphToString(
         map(RSAEncodeNum,
             StringToKgraph(text, k+1),
                key),
            k);
    end:
```

Now we try it out, with a 150-digit choice of n. Note that this means we'll be using 75-graphs, since $97^{75} < 10^{150} < 97^{76}$.

```
> keys:=RSAsetup(150):
  public:=keys[1]; private:=keys[2];
```

```
    public := [58275798824259224152729649923163918838517950830271585501 7\
    144484847421609272778490767597650118443597311849831459240406 7\
    384025550370142971709602674385 21, 1849982218625235549162928238 7\
    306972419512598470315009879478782685649065787593705898397921 6\
    6564841269310221397039646221986692531497073722912893849813]

   private := [58275798824259224152729649923163918838517950830271585501 7\
   144484847421609272778490767597650118443597311849831459240406 7\
   384025550370142971709602674385 21, 3311629069619390602835388907 7\
   810953835226409028594751957894266894819508519136812230877143 08\
   970811249627796735482001360890541195019272423373550779335 65]
```

```
> crypt:=RSAEncodeString(text,public);
```

```
      crypt := "?!2<*h&l9![</*(=X5'oMT\\*Mnc /T%j*r˜\"_=?b00'UP_\
      gQ#O¡oyVQdx3r—. N=s)¡!oidl?+$"
```

```
> RSADecodeString(crypt,private);
```

```
          "Once upon a midnight dreary, while I pondered, weak and weary"
```

12 RSA encoding a file

Now we will write a procedure that uses RSA to encrypt a file, rather than a string. Of course, we could just read in the string and use `RSAEncryptString`, but in some cases, it makes sense to use a different alphabet for the crypttext. For example, we might want to accept all possible ASCII codes 0-255 as plaintext input, but restrict our encrypted output to use a limited collection of characters (so that it can be printed or emailed without transcription problems, say). For example, in the example below, we use the character set of RFC 2045[35], which specifies how to encode base 64 MIME attachments for email.

[35] http://www.faqs.org/rfcs/rfc2045.html

Here is our procedure which reads the contents of a file, encrypts it with RSA, and writes out the result to another file.

```
> Alphabet:=cat(Select(IsAlphaNumeric,convert([seq(i,i=1..255)],bytes)),"+/");
```

```
> RSAEncodeFile:=proc(plainfile::string, cryptfile::string, key::list(posint))
    local  k, pf, cf, chunk, chunklen, codenum, crypt, Alen;
    global Alphabet;

    Alen    := length(Alphabet);
    chunklen:= ceil(log[Alen](key[1]));
    k       := floor(log[256](key[1]));

    try
      pf      := fopen(plainfile, READ, BINARY);
      cf      := fopen(cryptfile, WRITE, TEXT);
      chunk   := readbytes(pf, k);
      while (chunk <> 0) do
          codenum := map(RSAEncodeNum,
                         convert( chunk, base, 256, 256^k),
                         key);
          crypt:=KgraphToString(codenum, chunklen);
          writeline(cf, crypt);
          chunk := readbytes(pf, k);
      od;

    finally
       fclose(cryptfile);
       fclose(plainfile);
    end try;
    RETURN();
  end:
```

```
> RSADecodeFile:=proc(cryptfile::string, plainfile::string, key::list(posint))
    local  k, pf, cf, chunk, chunklen, plain, crypt, Alen;
    global Alphabet;

    Alen    := length(Alphabet);
    chunklen:= ceil(log[Alen](key[1]));
    k:=        floor(log[256](key[1]));

    try
      pf      := fopen(plainfile, WRITE, BINARY);
      cf      := fopen(cryptfile, READ, TEXT);
      chunk   := readline(cf);
      while (chunk <> 0) do
          plain:= convert(
                      map(RSAEncodeNum,
                          StringToKgraph(chunk, chunklen),
                          key),
                      base, 256^k, 256);
          writebytes(pf, plain);
```

```
            chunk := readline(cf);
          od;

        finally
           fclose(cryptfile);
           fclose(plainfile);
        end try;

        RETURN();
      end:
```

References

[DH76] W. Diffie and M. Hellman, “New Directions in Cryptography”, *IEEE Transactions on Information Theory* **IT-22** no.6 (1976), p. 644–654.

[Hil1] L. Hill, “Cryptology in an Algebraic Alphabet”, *American Mathematical Monthly* **36** (1929), p. 306–320.

[Hil2] L. Hill, “Concerning the Linear Transformation Apparatus in Cryptography”, *American Mathematical Monthly* **38** (1931), p. 135–154.

[HP] J. H. Humphreys & M. Y. Prest, *Numbers, Groups, and Codes*. Cambridge University Press, Cambridge. 1989.

[Kahn] D. Kahn, *The Codebreakers; The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Charles Scribner’s Sons, New York. 1996.

[Kob] N. Koblitz, *A Course in Number Theory and Cryptography* (Graduate Texts in Mathematics, No 114). Springer-Verlag, New York. 1994.

[NZM] I. Niven, H. Zuckerman, H. Montgomery, *An Introduction to the Theory of Numbers*, John Wiley & Sons, New York. 1991.

[RSA] R. L. Rivest, A. Shamir and L. Adelman, “A method for obtaining digital signatures and public-key cryptosystems”, *Communications of the ACM* **21** (1978), p. 120–126.

[Ros] K. Rosen, *Elementary Number Theory and its Applications*, 4th edition. Addison-Wesley, Boston. 2000.

[Sch] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd edition. John Wiley & Sons, New York. 1995.