Chapter 3

fsqFsHn sGGousG

3.1 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 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 "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 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!).

[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),

> 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.

3.2 Simple Ciphers

3.2.1 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.

As an example, let's implement this in Maple. Recall that we need to use a backquote (`) around character strings in Maple.[2]

```
> Alphabet :=`abcdefghijklmnopqrstuvwxyz `:
> Cryptabet:=`THEQUICKBROWNFXJMPSVLAZYDG `:
> Scramble:= msg ->
    cat(seq(substring(Cryptabet,
                      searchtext(substring(msg,i),
                                 Alphabet)),
            i=1..length(msg)));
```

$$Scramble := msg \rightarrow \mathrm{cat}(\mathrm{seq}(\mathrm{substring}(Cryptabet, \mathrm{searchtext}(\mathrm{substring}(msg, i), Alphabet)), i = 1..\mathrm{length}(msg)))$$

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

$$JWUTSU\ QX\ FXV\ PUTQ\ VKBS$$

Probably some comments are in order. First, we tell maple what our permutation is by defining the `Alphabet` that our plaintext is written in, and the corresponding `Cryptabet`. Our encoding function `Scramble` operates as follows (read from the middle out)

- `substring(msg,i)` extracts the i-th character from the input string `msg`.
- `searchtext` looks up where the character is in `Alphabet`.
- `substring` is used again, to extract the corresponding character from `Cryptabet`.
- this is done for each i ranging from 1 to the length of `msg`, giving a sequence of characters.
- finally, all the characters are smashed together again into one long string using `cat`.

Note that our "alphabet" contains a space, which corresponds to a space in our key. We didn't need to do this, but it left the message in words of the same length. This, of course, makes the message easier to guess. Also, notice that we used `searchtext`, which ignores case in the search (that is, "A" and "a" are the same). Had we wished to, we could have distinguished between upper an lower case using `SearchText` in which case we would also need to use a

[2] In Maple V release 5 and later, strings may be surrounded by double quotes ("), which is a more common way of denoting character strings. However, using a backquote works in all versions.

53 letter alphabet (and a 53 letter key). In this implementation, we also had to limit our input message to characters occurring in our alphabet. Had our message contained numbers or punctuation, the function `Scramble` would have failed with an error.

The usual way to break such a code is a combination of "frequency analysis" (that is, knowing that the most commonly occurring letter in the English language is "E", followed by "T"), plus a bit of trial and error. Leaving the spaces between words helps the cracker greatly.

3.2.2 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 (Caesar used 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.

Treating characters as numbers

Computers already have a built-in numeric representation of the alphabet— the ASCII encoding.[3] 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 digi**t**. A group of 8 bits is called a *byte*, and is a single "character" of information. A byte can have $2^8 = 256$ different states:

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

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.)[4] 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 letting the characters A–F stand for the integers 10–15. Thus, the byte 01011111 would be written as 5F in hexadecimal.

[3] 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.

[4] 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.

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, 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.[5]

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, if we like:

```
> seq([i,convert([i],bytes)],i=0..127);
```

[0,], [1, □], [2, □], [3, □], [4, □], [5, □], [6, □], [7, \a], [8, \b], [9, \t], [10, \n],
[11, \v], [12, \f], [13, \r], [14, □], [15, □], [16, □], [17, □], [18, □], [19, □], [20, □],
[21, □], [22, □], [23, □], [24, □], [25, □], [26, □], [27, \e], [28, □], [29, □], [30, □],
[31, □], [32,], [33, !], [34, \], [35, #], [36, $], [37, %], [38, &], [39, '], [40, (],
[41,)], [42, *], [43, +], [44, ,], [45, -], [46, .], [47, /], [48, 0], [49, 1], [50, 2],
[51, 3], [52, 4], [53, 5], [54, 6], [55, 7], [56, 8], [57, 9], [58, :], [59, ;], [60, <],
[61, =], [62, >], [63, ?], [64, @], [65, A], [66, B], [67, C], [68, D], [69, E], [70, F],
[71, G], [72, H], [73, I], [74, J], [75, K], [76, L], [77, M], [78, N], [79, O], [80, P],
[81, Q], [82, R], [83, S], [84, T], [85, U], [86, V], [87, W], [88, X], [89, Y], [90, Z],
[91, [], [92, \], [93,]], [94, ^], [95, _], [96, `], [97, a], [98, b], [99, c], [100, d],
[101, e], [102, f], [103, g], [104, h], [105, i], [106, j], [107, k], [108, l], [109, m], [110, n],
[111, o], [112, p], [113, q], [114, r], [115, s], [116, t], [117, u], [118, v], [119, w], [120, x],
[121, y], [122, z], [123, {], [124, |], [125, }], [126, ~], [127, □]

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 to its ASCII encoding, add a pre-specified offset modulo 127, then convert back to characters. It will look very similar to the `Scramble` function we wrote in §3.2.1, except we needn't declare an alphabet explicitly. However, in addition to the maple commands `convert( ,bytes)`, we will use another new command, `map`.

[5] 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.

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. Remember to read from the middle outward. Note that we are defining the function we are `map`ping directly.

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

$$Caesar := (plain,\ shift) \to \text{convert}(\text{map}((x,\ y) \to ((x + y - 1) \bmod 127) + 1,\ \text{convert}(plain,\ bytes),\ shift),\ bytes)$$

Let's give it a try:

```
> Caesar(‘Hello there‘,3);
```

Khoor#wkhuh

It seems to work. 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 afterwards 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.

Also, you might wonder why we can't use

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

in the middle of the function. If you try it, you'll see it does not work. This is because maple tries to fill in the value of `shift` at the wrong time.

Now, 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 □.[6] 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.

How can we get around this? We can restrict ourselves to printing characters using a variation of the idea in `Scramble` of §3.2.1, 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.

3.3 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);
```

$$5\,I$$

[6] 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.

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(‘Square root of ‘,x,‘is not a real number‘);
    fi;
  end;
```

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

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

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

```
> 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. [7]

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

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

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

What does that warning mean? It means that maple has assumed that the variable `y` used in `try` 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 `try(19)` (which sets *its* copy of `y` to 361), then check what the value of `y` is.

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

$$y := 35$$

```
> try(19);
```

$$25$$

```
> y;
```

$$35$$

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

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.

```
> try2:= proc(x)
    global y;

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

$$try2 := \mathbf{proc}(x)\,\mathbf{global}\,y;\ y := x^2\ ;\ \mathbf{if}\,25 < y\,\mathbf{then}\,25\,\mathbf{else}\,y\,\mathbf{fi}\,\mathbf{end}$$

```
> y;
```

$$35$$

```
> try2(19);
```

$$25$$

```
> y;
```

$$361$$

This time, of course, `y` was changed by invoking `try2`.

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. 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.

3.4 Caesar cipher redux

Now that we know about `proc`, lets implement the Caesar cipher again, this time with an arbitrary alphabet.

First, we define a global variable `Alphabet` that lists all the characters we intend to encode. Note that, in this case, the newline character is in our alphabet, as well as a space and a few punctuation characters.

```
> Alphabet := `ABCDEFGHIJKLMNOPQRSTUVWXYZ
  abcdefghijklmnopqrstuvwxyz., `;
```

$$Alphabet := ABCDEFGHIJKLMNOPQRSTUVWXYZ\backslash$$
$$abcdefghijklmnopqrstuvwxyz.,$$

Now, we can define two procedures, `ToNum` and `FromNum`, which convert character strings to and from lists of numbers. The numbers correspond to the position of the character in `Alphabet`, with (in this case) A=0. As in `Scramble` of §3.2.1, we use `substring` to pick out the i-th character, and `SearchText` to find its position in `Alphabet`. Note that `Alphabet` is declared as a global variable.

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

  FromNum := proc(numlist)
    local i;
    global Alphabet;
    cat(seq( substring(Alphabet,numlist[i]+1), i=1..nops(numlist)));
  end:
```

Now the Caesar cipher is written in terms of these functions. Note that this is a little easier to understand than the previous version.

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

    p       := length(Alphabet);
    textnum := ToNum(plaintext);
    codenum := [seq( modp(textnum[i]+shift, p),
                     i=1..length(plaintext)) ];
    FromNum(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 `ToNum` 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` (§3.2.2, p. 57). 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.

Let's see how it works.

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

YhqlBCYlglBCYlfl

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

$$test := I\ have\ heard\ the\ mermaids\ singing,\ each\ to\ each\backslash \\ I\ do\ not\ think\ that\ they\ will\ sing\ to\ me.$$

```
> encoded:=Caesar2(test,38);
```

$$encoded := tkQJdNkQNJ\backslash \\ MkbQNkVN\backslash \\ VJRMakaRWPRWPjkNJLQkbXkNJLQItkMXkWXbkbQRWTkbQJbkbQNgk\backslash \\ eRUUkaRWPkbXkVNi$$

Note that since our alphabet treats an end-of-line as a character, the linebreaks in the encoded version are significant. We can use the same function to decode, but giving the negative of the shift as the key.

```
> Caesar2(encoded,-38);
```

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

Finally, notice what happens if we try to encode characters that do not occur in our alphabet. To better see what happens, we will shorten the alphabet dramatically.

```
> Alphabet:=‘abcdefghijklmop_‘;
```

$$Alphabet := abcdefghijklmop_$$

Now let's re-encode the test message, 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(test,0);
```

$$_ha_e_hea_d__he_me_maid__i_gi_g__each__o_each__do__o__hi_k__ha__he__ill_i_g__o_me_$$

Can you explain why this happens?

3.5 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.

3.5.1 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[8], which was first used around 1600, and was popularly believed to be unbreakable. [9]

In our 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

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

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.

But, still, this is not very hard to crack. 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 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.

```
> Alphabet := `ABCDEFGHIJKLMNOPQRSTUVWXYZ
  abcdefghijklmnopqrstuvwxyz., `:

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

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

[8]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.

[9]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.

```
> coded:=Vignere(test,‘Prufrock‘);
```

$$coded := Yr\backslash$$
$$EkRcQuQjHrfINPbXVbNJMFrkMcTJWwquIQPIkGduIQPIIYrWSr\backslash$$
$$PbPi\backslash$$
$$McXcbxQlfiUFgPlaPaoTRAWuXdoNNN$$

We can make the decoding function from the original (let's call it `unVignere`) by changing exactly one + sign to a $-$.[10] 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.

```
> unVignere(code,‘Prufrock‘);
```

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

3.5.2 One-time pads

Note that the longer the key is in the Vignère cipher, the harder it is to break (provided the key-string isn't a well-known text, like the Gettysburg address or Hamlet's soliloquy). A *one-time pad* is essentially a Vignère cipher with an infinitely long, preferably random, key. Theoretically, this provides an unbreakable cryptosystem, because deciphering part of the message gives no information about the rest. The catch is that, to decode it, you must know what the infinitely long random sequence the message was encoded with.

Such systems were used by spies, where 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 is given his own codebook, and he uses one page per message.

A variation on this theme is the "Augustus cipher", 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.

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.[11] You might think that generating a random

[10] 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 `oMJYMPaT`: the numeric code of `P` plus the numeric code of `o` is 56 (the length of the alphabet), similarly for `r` and `U`, and so on.

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

sequence of numbers would be inherently unreproducible, so the message would be indecipherable unless we record the "random stream". 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.[12] 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 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`.[13]

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

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

It should probably also be mentioned that usually, a one-time pad is described by taking 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 really 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 for the version given above; we would need to modify the routine above to make a decryption proc by changing the $+$ to a $-$.

[12] 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. We shall ignore this problem here, however.

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

3.5.3 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.

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 $7 * 26 + 8 = 190$, assuming the usual correspondence of H=7, I=8. To convert back, we look at the quotient and remainder when dividing by 26. For example, $300 = 26 * 11 + 14$, yielding "LO".

Below is an example of how we might implement this conversion. We assume our usual functions `ToNum` and `FromNum` are defined, 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. It is assumed that the length of `text` is divisible by `k`.[14] A block of `k` characters $c_1c_2c_3\ldots c_k$ is assigned the numeric value $\sum_{i=0}^{k-1} x_i k^i$, where x_i is the value of `ToNum`(c_i). The only tricky part is getting the right characters in the right position.

```
> KgraphToKnums := proc(text, k)
    local textnums, i, j, p;
    global Alphabet;
    p:= length(Alphabet);

    textnums := ToNum(text);
    [seq( sum( textnums[(j-1)*k + i]* p^(k-i), i=1..k),
          j=1..length(text)/k)];
  end:
```

This may look a bit impenetrable at first, but when reading it, just keep in mind that `j` selects which `k`-block we are looking at, and `i` is the position of the character within that block.

To undo the transformation, we note that if

$$X = x_1 p^{k-1} + x_2 p^{k-2} + \ldots + x_{k-1} p + x_k,$$

then $x_i \equiv \left[X/p^{k-i}\right] \mod p^{k+1-i}$, where $[a]$ means the integer part of a (that is, we divide and discard any remainder).[15] We use maple's `iquo` function to do the division without remainder.

```
> KnumsToText := proc(numlist, k)
    local p;
    global Alphabet;
    p:=length(Alphabet);
    FromNum([seq(seq(iquo(modp(numlist[j], p^(k-i+1)),
                          p^(k-i)),
                     i=1..k),
                 j=1..nops(numlist))]);
  end:
```

[14] See the routine `TextToVects` in §3.8.1 for an way around this

[15] There are more computationally efficient ways of doing this, but we aren't worrying about that.

In the examples below, we are using the 26-character alphabet discussed above. Of course, this will work on any alphabet.

```
> KgraphToBignums(`HILOHOLA`,2);
```

[190, 300, 196, 286]

We can also use 4-graphs:

```
> KgraphToBignums(`HILOHOLA`,4);
```

[128740, 132782]

```
> KnumsToText([128740, 132782],4);
```

HILOHOLA

Another way to treat multiple characters together is to think of them as vectors. For example, the digraph "HI" might correspond to the vector [7, 8]. We will treat this approach in §3.8.

3.6 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 §3.5.2 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.[16]. 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.[17] Below is a small example of a procedure which encrypts a file one line at a time using a procedure `EncryptLine`. It should be clear how to modify it to use your favorite encryption routine instead.

[16] 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`

[17] On several operating systems, including MS-Windows and MacOS (but not Unix), 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, no translation is necessary, but it doesn't hurt.

```
> 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:
```

3.7 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 §3.5 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 $x \mapsto ax + b \pmod{p}$ 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.

3.7.1 When do affine encodings fail?

You may wonder, however, will this work? 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 3.7.1 *Let a, b, x, and y be integers in the set $\{0, 1, \ldots, p-1\}$, with $\gcd(a, p) = 1$. Then*

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

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 we can include all the usual printable ASCII characters (see §3.2.2, especially the table on page 56). The keyboard I am typing this on produces 96 distinct characters (including space, tab, and newline), without resorting to control keys. 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.

3.7.2 Implementing and using an affine encoding

Implementing this scheme is a trivial modification to the `Caesar2` procedure we wrote in §3.4. We only need add another parameter `a`, and change the encoding calculation. (We also renamed `shift` as `b` to correspond to our discussion here).

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

    p       := length(Alphabet);
    textnum := ToNum(plaintext);
    codenum := [seq( modp(a*textnum[i]+b, p), i=1..length(plaintext)) ];
    FromNum(codenum);
  end:
```

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.

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 the group $0, 1, \ldots, p-1$. (As before, we can run into trouble here if a and p are not relatively prime.) Fortunately for us, maple knows about division modulo p:

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

$$81$$

3.7.3 Breaking an affine cipher

Suppose we have a message that was encrypted with an affine cipher on an alphabet we also know. 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 we know the message was enciphered with $x \mapsto y = (ax + b) \bmod p$, and we know 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 can ask maple to do this for us using `msolve`:

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

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

but, it will be more convenient to solve the *deciphering* transformation instead:

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

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

```
> AffineCode(‘nrIbQyxEbkrI bwr EUbJaPybL abpyJJy UbUlrrxU‘,44,13);
```

You bend your words like Uri Gellers spoons

3.8 Enciphering matrices

3.8.1 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 §3.5.3, we could think of it as a vector of characters. Thus, if we were using, say, 2-vectors, then the name "Dougal" might become the three vectors [36, 79], [85, 71] and [65, 76]. If the length of the text we were converting was not divisible by 2, we would have to extend it by

adding a space. Thus the name "Ian" occurring at the end of a message would become the pair of vectors [41, 65] and [78, 0].

Below is an implementation of this conversion, `TextToVects`, which takes as input the text to convert to n-vectors and the length of the vectors, n. It uses the `ToNum` procedure from §3.4, as well as the usual maple commands.

```
> TextToVects:= proc(text,n)
    local i,j,textnums,vectors;

    textnums := ToNum(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.

Converting back from a list of n-vectors to a character string is easy, because we already have `FromNum` to convert a list of numbers to a character string for us. All we need to do to change a list of lists to just a list is discard the inner sets of braces. 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 maple distinguishes 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. So, to be sure that `op` will work, we need to convert the vectors to lists first. The result is

```
> VectsToText:= vects -> FromNum(map(op, map(convert,vects,list))):
```

3.8.2 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 (§3.5.1) 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.

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".

In maple, one does matrix arithmetic pretty much as you would expect, except that you need to use `&*` to indicate matrix multiplication. This is because matrix multiplication is not commutative, while `*` is. Also, maple doesn't carry out the operations until you execute `evalm`. Thus, to compute $Av + b$, we would write `evalm(A &* v + b)`.

```
> AffineMatEncode := proc(text,A,B)
  local   vtext,vcrypt,i,n,p;
  global  Alphabet;

  p:=length(Alphabet);
  n:=nops(convert(B,list));

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

Note that matricies 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 $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$:

```
> AffineMatEncode('I'm so mixed up!', [[0,1],[1,0]], [0,0]);
```

'I mosm xideu !p

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([[67, 23], [69, 81]]); B:=[5, 47];
```

$$A := \begin{bmatrix} 67 & 23 \\ 69 & 81 \end{bmatrix}$$

$$B := [5, 47]$$

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

nG=a/$TvPuUU

```
> AffineMatEncode(‘nG=a/$TvPuUU‘, inverse(A), evalm(-inverse(A) &* B));
```

Lester Hill

3.8.3 A Known-plaintext attack on an affine matrix cipher

As in §3.7.3, 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 $n+1$ 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 crypttext "oHyeAH". This is sufficient information to decipher the message, because we know the encodings of three different 2-vectors.

```
> plain:=TextToVects(‘secret‘,2);
```

$$plain := [[83, 69], [67, 82], [69, 84]]$$

```
> crypt:=TextToVects(‘oHyeAH‘,2);
```

$$crypt := [[79, 40], [89, 69], [33, 40]]$$

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.[18]

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];
```

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

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

[18] You probably want to check that this works for yourself.

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]);
```

$$[79\,a + 40\,b - 83 + e,\ 79\,c + 40\,d - 69 + f]$$

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)};
```

$$eqns := \{33\,a + 40\,b - 69 + e,\ 33\,c + 40\,d - 84 + f,\ 79\,a + 40\,b - 83 + e,$$
$$79\,c + 40\,d - 69 + f,\ 89\,a + 69\,b - 67 + e,\ 89\,c + 69\,d - 82 + f\}$$

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

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

$$Answer := [\begin{bmatrix} 72 & 85 \\ 6 & 82 \end{bmatrix}, [16,\ 1]]$$

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.

3.9 Modern cryptography

3.9.1 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. At first, this may seem like an unreasonably strong definition; you may be thinking that, given a mass of encrypted data, how is the cryptanalyst to know *how* it was encrypted? If you are encrypting your own files, for personal use, this is probably so. 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 as a good measure of security.

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[19]).

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.[20] 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.

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 during 1995 in about 8 days using 112 networked computers (see Damien Doligez's web page about cracking SSL[21] for more details). Even though this algorithm has been "cracked", it seems that this scheme is certainly good enough for everyday use considering the resources required to decrypt a single session. In any case, current web browsers offer much more security— a 40-bit key is the weakest offered.

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

3.9.2 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 message with the same hash.

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

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).

3.9.3 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 (§3.7) $x \mapsto ax + b \bmod p$, if we know the value of a and b to used encode the message, we can quickly compute the deciphering key $1/a \bmod p, -b/a \bmod p$. Also, decrypting the message usually also yields the key.[22]

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 W. Diffie and M. Hellman invented "public key cryptography". 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).

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.[23] However, if the merchant provides a public key,

[22] A story related to this appears in Casanova's memoirs (1757). He was given an encrypted manuscript by a Madame d'Urfé, which he deciphered. Later, upon learning this, Madame d'Urfé was incredulous, believing he must know the keyword to do so.

> 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 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.

[23] 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 and 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 it is still quite a lot. 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 microwave

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.[24]

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 §3.9.2) computed on its contents. This feature is commonly used to "digitally sign" email by software such as PGP[25].

A common public key system in current use is RSA (named for its inventors: Rivest, Shamir and Adelman), which we will look at in detail in §3.11. 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.

3.10 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 for more information.

3.10.1 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,

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

receiver.

[24] 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.

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

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 30 minutes on a 133 MHZ pentium) 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 $\gcd(138, 126)$,

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

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

Theorem 3.10.1 (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:*

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

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

Before proving Thm. 3.10.1, we will need a lemma.

Lemma 3.10.3 *Let a be a positive integer, and b be a nonnegative integer. If*

$$b = aq + r, \qquad \text{with } q \text{ and } r \text{ positive integers}$$

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

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 $b - aq$. Since $r = b - aq$, we have d a divisor of r. So it must divide $\gcd(a, r)$.

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

Thus, $\gcd(a, r) = \gcd(a, b)$. □

Now we are ready to prove Thm. 3.10.1:

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, $k_{n+1}r_n = r_{n-1}$ says that r_n is a divisor of r_{n-1}, so certainly $r_n = \gcd(r_n, r_{n-1})$. Applying the lemma to the equation above it gives $\gcd(r_n, r_{n-1}) = \gcd(r_{n-1}, r_{n-2})$. We can construct the following chain in this way:

$$r_n = \gcd(r_n, r_{n-1}) = \gcd(r_{n-1}, r_{n-2}) = \ldots = \gcd(r_1, b) = \gcd(a, b)$$

□

Recall that if r is such that $ar = 1$, then r is called the multiplicative inverse of a. 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. 3.7.1 slightly reworded.

Theorem 3.10.4 *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

$$ak \equiv 1 \bmod n.$$

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 $\gcd(a, n) = 1$.

Now suppose that $\gcd(a, n) = 1$. Then by Cor. 3.10.2, there are integers r and s with $ar + ns = 1$. This means $ar - 1$ is divisible by n, so

$$ar \equiv 1 \bmod n.$$

But this says that r is the inverse of $a \bmod n$. □

Notation. We haven't yet used this notation, but it is common 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 have already know, this set is closed under addition and multiplication, and the usual associative, distributive, and commutativity laws hold. In fact, this set forms a *ring*.

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

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

3.10.2 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 3.10.5 *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 $mn - 1$.

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:

$$c \equiv ant \equiv a(1 - mk) \equiv a \mod m$$

and

$$c \equiv bkm \equiv b(1 - nt) \equiv b \mod n.$$

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

3.10.3 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 4 numbers $3, 9, 7, 1$ and then it repeats. Now consider the powers of 11:

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

This sequence is periodic of period 2. What about 4? We have

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

This is periodic, but the sequence does not contain 1. Finally, let us consider the powers of 6:

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

The first two examples give rise to the following definition:

Definition 3.10.6 *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 3.10.7 *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 $[\![a^{k-1}]\!]_n$. By Thm. 3.10.4, 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

$$a, [\![a^2]\!]_n, [\![a^3]\!]_n, \ldots, [\![a^n]\!]_n, [\![a^{n+1}]\!]_n.$$

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 $[\![a^k]\!]_n$ and $[\![a^t]\!]_n$, with $1 \leq t < k$. Since

$$[\![a^k]\!]_n = [\![a^t]\!]_n$$

we have

$$\left[\!\left[a^{k-t}\right]\!\right]_n = 1.$$

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 3.10.8 (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.

3.10.4 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 mod n.*

Definition 3.10.9 *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,[26] *or sometimes Euler's totient function.*

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

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

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

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 3.10.10 *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. 3.10.5), 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 3.10.3. 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 $r = \llbracket t \rrbracket_a$. 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 $s = \llbracket t \rrbracket_b$, 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 3.10.11 (Euler, 1750) *Let a and n be relatively prime integers, with $n \geq 2$. Then*

$$\llbracket a^{\varphi(n)} \rrbracket_n = 1.$$

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

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

Since a and n are relatively prime, $\left[\!\left[a\right]\!\right]_n \in \mathbb{Z}_n^*$. 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 $\left[\!\left[ab\right]\!\right]_n = \left[\!\left[ac\right]\!\right]_n$ if and only if $\left[\!\left[b\right]\!\right]_n = \left[\!\left[c\right]\!\right]_n$. 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 $\left[\!\left[N\right]\!\right]_n \in \mathbb{Z}_n^*$. 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

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

Thus,

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

□

3.11 The RSA Public key cryptosystem

In 1978, R. Rivest, A. Shamir, and L. Adelman published the description of the RSA public key system, which we will describe here. As discussed in §3.9.3, 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.

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 large number $a < \min(p,q)$, called the **exponent**, which is relatively prime to $\varphi(n)$. We make the numbers (n,a) public— these form the key needed for encoding. We also use the Euclidean algorithm to compute x which satisfies $ax - y\varphi(n) = 1$; that is, so that $\left[\!\left[ax\right]\!\right]_{\varphi(n)} = 1$. The number x is the part of the key we keep private.

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 §3.5.3, with $0 \le \beta < \min(p,q)$. For each block of plaintext, the sender transmits $\left[\!\left[\beta^a\right]\!\right]_n = m$.

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

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

This relies on Euler's theorem, so that $\left[\!\left[\gamma^{\varphi(n)}\right]\!\right]_n = 1$, where $\gamma = \beta^y$. The requirement that β be less than both p and q was to ensure that $\gcd(\beta, n) = 1$, so that we can apply the theorem.

Note that the symmetry between a and x means that we can make either one public. Thus, RSA works just as well for authentication as it does for public key encryption.

3.11.1 Implementing RSA in Maple

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

3.11.2 Useful preliminaries

First, we define a procedure which allows us to read in a file, and convert it into a list of large integers. This is done using `readbytes` to obtain `chunksize` bytes from the file. Each byte is represented by its ASCII code[27] (that is, as an integer between 0 and 128), and each group of `chunksize` characters is viewed as a number expressed in base $128^{\texttt{chunksize}}$. We convert each of these numbers to decimal, as in §3.5.3. The result is a list of several rather large integers representing the contents of the file. The choice of `chunksize` is fairly arbitrary, provided you ensure that $128^{\texttt{chunksize}}$ is less than your chosen primes. Here we use 47, which gives us 99 digit numbers, because 128^{47} is a 99-digit number.

```
> ReadBignums:= proc(filename,chunksize)
    local file,chunk,i,p,numlist;

    p       := 128;
    numlist := [];
    file    := fopen(filename, READ, BINARY);
    chunk   := readbytes(file,chunksize);
    while (chunk <> 0) do
        numlist := [op(numlist),
                    sum( chunk[i]* p^(chunksize-i), i=1..nops(chunk))];
        chunk := readbytes(file,chunksize);
    od;
    close(file);
    RETURN(numlist);
  end:
```

For example, if we read in the contents of the file `/home/mat331/data/Extra` using 47-byte chunks, we obtain a list of 39 numbers of 99 digits each.

```
> data:=ReadBignums(`/home/mat331/data/Extra`,47):
> nops(data);
```


[27] We assume here that the data in the file is plain text. If it is expected that there may be extended ASCII codes in the file (or if the file contains "binary data", for example, if it is compiled computer code or an image), the variable `p` should be set to 256 instead.

```
> data[2];
```

$$882221479087147858856410399353400203005463343834450424221849495430681\backslash$$
$$46627358316365192956608760830695$$

It will also be useful for us to be able to convert our big integers back into text, so that we can read our message when we are done.[28]

```
> BignumToText:=proc(num,k)
    local i,p;
    p:=128;
    convert([seq(  iquo(modp(num,p^(k-i+1)),p^(k-i)), i=1..k)],'bytes');
  end:
> BignumToText(data[2],47);
```

g phrase, which was encoded using a affine
enco*

Basic Setup: choice of primes, base, exponent, and decoding exponent

We wish to choose two primes p and q of about 100 digits each "at random", and also a random exponent relatively prime to $\varphi(pq)$. At first, this may seem to be a very hard task: how do you know if an arbitrary 100-digit number is prime unless you try to factor it? However, there are a number of efficient ways to test if a number is prime without actually determining its factors. Maple's `isprime` function uses such methods. Thus, all we really need is to generate some random numbers of about 100 digits, then take the next prime bigger. We use Maple's `rand` to create two functions, `BigRand` and `MedRand`, which give us random numbers of around 100 digits and less than 90 digits, respectively. We also have to remember to use `randomize()` to ensure that we get a different random number each time (see the discussion of pseudo-random numbers in §3.5.2, page 64).

```
> BigRand := rand(10^100..10^105):
  MedRand := rand(50..10^90):
> readlib(randomize)();
```

892060192

Now we pick our primes p and q, set `phi` to be $\varphi(pq)$, and compute our base $n = pq$.

```
> p:=nextprime(BigRand());
  q:=nextprime(BigRand());
  phi:=(p-1)*(q-1);
  n:=p*q;
```

$$p := 168148344694840258566703067925059488643059382079284570104453791\backslash$$
$$3863119894063560618824877602199869749797 31$$

$$q := 759147509371863656660616161981274323356040723687316036705747045\backslash$$
$$9407203590662948347772517936413042598626 47$$

[28] Of course, if you anticipate 8-bit data in your input, the number `p` here will need changing, as well.

$$\phi := 12764939708008960576348156136037597347167169331860339503010769812699154114822845932599138076073100856701277725767099009433810307591820570871142464403355472591199517479954681219469395759024153764817168\\8434165580$$

$$n := 12764939708008960576348156136037597347167169331860339503010769812699154114822845932599138076073100856701370455352505679825333039514811204252342374413932132651880537563687384454316660848690127720203297\\9669007957$$

To choose our public exponent a, we pick a number at random and check if it is relatively prime to $\varphi(n)$. If not, we just pick another and try again. Usually, a suitable exponent will be found after only a few tries.

```
> a:=MedRand();
  while( gcd(a,phi) <> 1) do;
      a:= MedRand();
  od;
```

$$a := 23761842762862818026813705680848933572643894137172039301368795022049395663850363780640313 5$$

$$a := 31980669092497318435254308131511089664020984689213578921745876063910817576939119242719832 0$$

$$a := 31313498486294424311286178163628183559546522196716411592492267847892755233552248294501656 7$$

Finally, we choose x to be the inverse of a mod $\varphi(n)$.

```
> x:= modp( 1/a, phi);
```

$$x := 16445856393380707369282446076185129880820098375956674147069759642064503707641544915321985624002512316665365378468661955298331280883922977475968127108659879154551620218885868380952053297001254368234914\\293234343$$

Encoding and Decoding

We've already done all the hard work; now writing the actual encoding and decoding routines is easy. Lets write `RSAEncodeNum` which encodes a single large number β by computing $\beta^n \bmod n$.

```
> RSAEncodeNum := proc(bignum, a, n)
    modp( bignum &^ a, n);
  end:
> RSAEncodeNum(data[1],a,n);
```

```
650640533631773130566525918154403948990214682783764008970814724574 98\
    95141947823205579680281872956674957884890678666592759764003894389372\
    05499067528982324528262147573016171219176997233479109398560183821065\
    60214
```

We can easily write a one-liner to encode a file with RSA by using `map` to apply `RSAEncodeNum` to the result of `ReadBignums`.

```
> RSAEncodeFile := proc(file,n,a)
    map(RSAEncodeNum, ReadBignums(file,47), a, n)
  end:
> encoded := RSAEncodeFile(‘/home/mat331/data/Extra‘, a, n):
```

To check that it worked, we can decode the message we just encoded by using `RSAEncodeNum` with the inverse of `a` that we computed earlier, and then apply `BignumToText` so we can read the result.

```
> decoded := map(RSAEncodeNum, encoded, x, n):
> cat(op(map(BignumToText,decoded,47)));
```

Here is exercise #3.

[1.] Decode the following phrase, which was encoded using a affine encoding cipher on a 27 − letter alphabet (the letters a − z and a blank). Note that a = 0.

xgb dkjvmbkffcmtkrv m lkaerask mdbmwkfmezremzkmxrcm kkmfvkrx mxgvkmaskrvsc