Oxford University Computing Laboratory

MSc in Computer Science

Functional Cryptography

Using a functional language to implement some cryptographic algorithms

Stephen Drape

September 2001

Contents

Contents

Chapter 1 Introduction

The aims of this project are listed below:

- To give an overview of cryptography by examining some cryptographic algorithms.
- To look at some of the techniques used in both creating and breaking ciphers.
- To briefly consider some of the issues associated with different algorithms such as security, ease of implementation and efficiency.
- To present applications of functional programming.
- To exploit the style of functional programming and try to create efficient implementations.

The main areas of this project are **Cryptography** and **Functional Programming**. The following sections will look at these two topics.

1.1 Cryptography

Cryptography is the study of the ways in which the contents of a message can be disguised. The main aim of cryptography is to mask a message in such a way that it would be unreadable to anyone intercepting the transmission of the message.

1.1.1 Some terms and definitions

The process of disguising a message will be called **encryption** (also known as enciphering or encoding) and recovering the contents of a message will be called **decryption** (deciphering or decoding). The original message is called the **plaintext** and after encryption it is known as the **ciphertext**. The ciphertext can consist of a series of letters or numbers - depending on the algorithm used. The algorithms in this project used to encrypt messages are examples of *ciphers*. A **cipher** replaces each letter in the plaintext by another letter or a number, thus creating the ciphertext (by contrast, a **code** usually replaces a whole word or phrase with a character). A cipher usually makes use of a *key*. A **key** is a sequence of characters (either letters or numbers) which encrypts a plaintext in a particular way. **Cracking** or **breaking** describes the process of recovering a plaintext without the knowledge of a key.

1.1.2 Ciphers

A cipher [Wsh] can be thought as having a function e, which takes a plaintext and produces a ciphertext. If M is the plaintext, C the ciphertext and k the key then $C = e(M, k)$. There should also be a decryption function d such that, for some key k' (which could be the same as k), $M = d(C, k') = d(e(M, k), k')$. Any cryptographic algorithm which is developed should satisfy the latter property.

1.1.3 The importance of cryptography

Cryptography is used in a wide range of areas from smart cards to military communications. The use of cryptography has been important throughout history - an example is the cracking of the Enigma ciphers during the Second World War [Sgh]. However, with the increased use of technology such as mobile phones and the Internet, cryptographic algorithms need to be applied quickly and efficiently and they should guarantee a high level of security.

1.1.4 Choosing ciphers

The ciphers which are implemented in this project have been picked for a number of reasons. All of the ciphers chosen are easy to understand and do not rely on any deep knowledge of cryptography. The earlier ciphers will be used to show how ciphers may be broken, and the latter ciphers will highlight the reliance that modern cryptographic algorithms have on "hard" Mathematical problems. Both encryption and decryption functions will be developed for each cipher. The encryption function will be applied to text strings and its ciphertext could consist of another text string or a sequence of integers. Correspondingly, the decryption function will take in a ciphertext and produce a text string.

1.2 Functional Programming

Most implementations of cryptographic algorithms use imperative languages. However, this project will use a functional language instead - the language that will be used is called **Haskell**. (Most of this section refers to [Brd].)

1.2.1 The functional style of programming

A **functional** program is built from expressions (the *functions*). A program is executed by evaluating an expression. Expressions may be defined using standard functions, usually contained in the **Prelude**. Associated with each expression is its **type** - this determines what the expression takes in as arguments and what the output should be. The first line of function usually gives its type (although it is not required by Haskell) and is of the form `function:: a -> b`. For example, a function which calculates the length of a string should take in a string as the input and produce an integer as the output. Some example types are `Int` (limited precision integers), `Integer` (arbitrary size of integers), `Char` (characters), `String` (list of characters), `Bool` (`True` or `False` values), `[a]` (lists of type `a`). The last type is an example of a *polymorphic type*.

1.2.2 Lists

One of the most useful types in Haskell is the *list*. A list is built up by using `:` . The empty list is denoted by []. For example `1:2:3:[]` is a list with 1 as the first element (called the **head** of the list), followed by 2 and the last element is 3 - in Haskell, `[1,2,3]` can be used as a shorthand for this list. The usual convention for a list is of the form `x:xs`, where `x` is the head of the list and `xs` is the rest of the list (called the **tail**). A recursive function can therefore be defined easily on lists by an expression of the form:

```
f []     = e
f (x:xs) = (g x) : (f xs)
```

[`f` has type `[a] -> b` `g` has type `a -> b` `e` has type `b`.]

Some useful prelude functions for lists:

`take n xs`	displays the first *n* elements of *xs*.
`drop n xs`	removes the first *n* elements of *xs* and shows the remainder of the list.
`map f xs`	applies the function *f* to each element of *xs*.
`filter t xs`	uses a Boolean test *t* and prints the elements of *xs* that satisfy the test.

`xs !! n`	produces the *n*th element of the list (counting starts at 0).
`[a..b]`	the list of integers from *a* to *b* inclusive (also used with characters).
`xs ++ ys`	joins two lists together.
`length xs`	shows the length of the list.
`reverse xs`	reverse the order of the list.
`zip xs ys`	makes a list of pairs (*a*,*b*) where *a* is from *xs* and *b* from *ys*.

`foldr` is a function which can represent a recursion on lists:

```
foldr           :: (a -> b -> b) -> b -> [a] -> b
foldr f e []    = e
foldr f e (x:xs) = f x (foldr f e xs)
```

[`foldr1`, `foldl` and `foldl1` are similarly defined.]

Strings are also lists, so `String = [Char]`. `"abc"` is a shorthand for `'a':'b':'c':[]`

1.2.3 Some other Prelude functions

- Functions are needed to convert between characters and their ASCII values.
`chr` takes an ASCII values and returns the corresponding character and `ord` returns the ASCII values for a given character. The types of these functions are:

```
ord ::  Char -> Int
```
and `chr:: Int -> Char`

- `fst` and `snd` are used with pairs: `fst(a,b)=a` and `snd(a,b)=b`

- `mod x d` works out *x* (mod *d*) and `rem x d` works out the remainder when *x* is divided by *d*. When *x* and *d* are both positive, then `mod` and `rem` give the same values. However, they can give different answers when one of the numbers is negative. In most cases, `rem` will be used as it is generally requires less reduction steps. Corresponding, the quotient can be given by using `div` or `quot` and again these agree for positive numbers. Also useful will be:
`divMod x d = (div x d, mod x d)` and `quotRem x d = (quot x d, rem x d)`

- A function can be specified by using the following notation

```
\ variables -> expression
```

e.g., `\a b -> a * b`
defines a function which takes two values and returns their product. This notation saves writing a separate function for something which may be only used once (i.e. a local definition).

1.2.4 Why use functional programming?

Functional programming is rarely used with cryptography as imperative languages are considered faster and use “traditional” programming concepts such as arrays, pointers and modules. However, functional programming does have several advantages, some are which are highlighted below.

- Functional programs are built up function by function. This means that a complicated problem can be simplified by breaking up the problem and then writing a function for each small step. Once a function is created, it can often

be adapted easily for other programs. Creating efficient programs can be achieved by concentrating on individual functions instead of looking at the program as a whole.

- Functional programs are quite easy to test and debug. Functions can be tested individually. This means that the places where errors occur can be spotted quickly and the offending function can be changed. In the implementation of each cipher, sample output from many functions will be shown.
- Functionality can be utilised to assist with proofs for correctness. In imperative languages, concepts such as invariants have to be used to prove correctness.
- Haskell is a lazy functional language. This means that results are only worked out on demand. Laziness can be exploited in programs. For instance, consider the functions:
```
list = map (^10) [2..100]
first = head list
```
To evaluate `first`, only the head of the list worked out as the rest of `list` is not needed. An imperative program would have to work out the whole list and then just return the first element.
- Most of the cryptographic algorithms used are based on mathematical functions. These should be easier to create in a functional environment, as the language is designed to deal with functions.

1.2.5 Hugs

Most of the programs in this project are executed by Hugs (Haskell Users Gofer System). Listings of the Haskell files are contained in Appendix A - all the files are "literate" (with extension .lhs). In a **literate** file, all the lines of code are prefixed by ">" and then comments can be added in between the pieces of code. The files were created and executed using a remote login to the Sun machines in the Engineering Department. Any timings that are given have been produced using these machines.

When giving sample output from functions, the number of reduction steps needed and the number of cells used are shown. (In complicated calculations, the number of garbage collections is also displayed). These numbers give an indication of how many steps have to be performed to evaluate the expression and how much space was consumed. These are not accurate measures of complexity or the time taken although they are a useful guide to help with deriving efficient programs.

1.3 The Focus

The focus of the project is to implement cryptographic algorithms using Haskell. During the implementation, issues relating to both cryptography and functional programming can be discussed. The main purpose is to present cryptography from a new angle - namely by studying functional programming. During the course of the project, insight into both cryptography *and* functional programming should be gained.

This project builds on work studied in the courses:

- *Functional Programming*
- *Advanced Functional Programming*
- *Procedural Programming*.

Chapter 2 The Caesar Cipher

The Caesar cipher is one of the simplest ciphers known. It is easy to apply but very insecure. It is useful to see this cipher as an example for discussing the techniques used in making encryption and decryption algorithms. One way of breaking the Caesar cipher uses a technique known as ***frequency analysis***. The purpose of this chapter is to implement a functional program which can encipher texts using the Caesar cipher. Also, a cracking program is to be developed - the aim of this program is to produce an algorithm that displays the most likely plaintext after applying a series of tests.

2.1 What is the Caesar cipher?

The Caesar cipher [Sgh, Wsh] acts on a string of letters by transposing each letter of the plaintext by a given number of places - this defines the key. The key can either be a number (between 0 and 25) or a letter (A represents 0, B represents 1, etc.).

For example:

Suppose the key is 12

Plaintext	`THE HOUND OF THE BASKERVILLES`
Ciphertext	`FTQ TAGZP AR FTQ NMEWQDHUXXQE`

To encipher each letter, each letter is shifted the appropriate number of places right along the alphabet cycling where necessary. It is often useful to write down the alphabet with the shifted alphabet underneath.

So, for a shift of 12

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
M N O P Q R S T U V W X Y Z A B C D E F G H I J K L
```

When using the Caesar cipher, the cipher is only applied to uppercase letters, leaving the blanks alone. To achieve this, all lowercase letters need to be converted to uppercase and any symbols (such as numbers, +, &, .) need to be removed or ignored.

To decipher, it is necessary to shift the alphabet back (or forward) by the correct number of places. So for example, to decipher a message with a key of 12, either shift the alphabet to the left by 12 places or right by 14 places. So the same algorithm can be used to encipher and decipher a message. In fact, if $c(s, m)$ represents a Caesar cipher with shift s applied to a message m, then

$c(-s, c(s, m)) = m$ (assuming m just contains uppercase letters and blanks)

2.1.1 The enciphering part

To use a Caesar cipher, it is necessary to change all the lowercase letters to upper letter. To do this, first define a function *capitalise*:

```
> capitalise :: Char -> Char
> capitalise c = if isLower c then chr (off + ord c) else c
>       where off = ord 'A' - ord 'a'
```

This function makes use of the Prelude boolean test `isLower` to test whether a character is a lowercase letter.

To use this in a string (remembering `String = [Char]`):

```
> caps :: String -> String
> caps = map capitalise
```

Next, it is necessary to remove all the unwanted characters. To do this, a boolean test `isWanted` is created which tests whether a character is an uppercase letter or a blank. `isWanted` can then be used with `filter` for use on a string.

```
> isWanted :: Char -> Bool
> isWanted c = (c==' ')||(isUpper c)
```

This uses another Prelude test called `isUpper`.

To shift each letter according to the key, it is necessary to define a function which changes each uppercase letter (cycling where necessary) and leaves blanks unchanged:

```
> nlett :: Int -> Char -> Char
> nlett n c
>   | c>='A' && c<='Z' = chr(ord 'A' + (mod(ord c - ord 'A' + n) 26))
>   | otherwise        = c
```

To work on a string, it will be necessary to use `map`.

Putting all these parts together gives a definition for `caesar`:

```
> caesar :: Int -> String -> String
> caesar n s = map (nlett n) (filter isWanted (caps s))
```

Some examples using these functions:

```
Main> caps "abcdE f"
"ABCDE F"
(310 reductions, 397 cells)
Main> nlett 3 'G'
'J'
(62 reductions, 97 cells)
Main> caesar 1 "h a l 4*7"
"I B M "
(547 reductions, 707 cells)
```

Now, using the plaintext in Section 2.1:

```
Main> caesar 12 "THE HOUND OF THE BASKERVILLES"
"FTQ TAGZP AR FTQ NMEWQDHUXXQE"
(2377 reductions, 3107 cells)

Main> caesar (-12) "FTQ TAGZP AR FTQ NMEWQDHUXXQE"
"THE HOUND OF THE BASKERVILLES"
(2377 reductions, 3107 cells)
```

2.1.2 Weaknesses of the Caesar cipher

The Caesar cipher is a very elementary cryptographic algorithm. It is quick to apply

and the key that is needed is only a small number (or a letter). Having such a simple key means that the encryption is weak and once the key is known, it is very easy to decrypt a Caesar message. One way of finding the key relies on a method known as *frequency analysis*.

2.2 Frequency analysis

English (as with all languages) uses certain letters more than others. By studying the frequency of letters in the ciphertext, it is possible to guess what some of the letters in the ciphertext represent. Just finding what one letter in the ciphertext represents is enough to find the whole plaintext. So, frequency analysis is a very powerful tool for this cipher.

For English, the six most common letters in are:

E T A I N O

(A table of letter frequencies is printed in [Wsh].)

2.2.1 Functional frequency analysis

The frequency of a particular character is computed by counting up the number of occurrences of that character in the ciphertext – this can be achieved functionally by using `filter` and then `length`. The function `freq` takes in a list of letters and a ciphertext and creates a list of pairs of the form (*frequency*, *letter*). The reason for creating the list in this way is so that the letter with the highest frequency can be quickly identified. The function `analyse` can be changed to check for characters other than uppercase letters.

```
> freq :: String -> String -> [(Int,Char)]
> freq [] _ = []
> freq (l:ls) xs = (length (filter (==l) xs),l): (freq ls xs)

> analyse :: String -> [(Int,Char)]
> analyse = (freq ['A'..'Z']).caps
```

For example:

```
Main> analyse "the quick brown fox jumped over the lazy sleeping dog"
[(1,'A'),(1,'B'),(1,'C'),(2,'D'),(6,'E'),(1,'F'),(2,'G'),(2,'H'),(2,'
I'),(1,'J'),(1,'K'),(2,'L'),(1,'M'),(2,'N'),(4,'O'),(2,'P'),(1,'Q'),(
2,'R'),(1,'S'),(2,'T'),(2,'U'),(1,'V'),(1,'W'),(1,'X'),(1,'Y'),(1,'Z'
)]
(6856 reductions, 6925 cells)
```

From this example, it can be seen that every letter occurs at least once. The fifth element `(6,'E')` has the highest first component - this means that the letter “E” has the highest frequency (which equals 6).

It is now necessary to give the letter with the maximal value (if two letters have maximal values, then the leftmost letter is given)

First define a function which finds the maximum first component of a list of pairs:

```
> maxpair :: [(Int,a)] -> (Int,a)
> maxpair = foldr1 (maxfst)
```

where:

```
> maxfst :: Ord e => (e,f) -> (e,f) -> (e,f)
> maxfst (a,b) (c,d) = if a>=c then (a,b) else (c,d)
```

Now use `maxfst` to find the letter (the second component) for which this maximum occurs:

```
> maxlet :: String -> Char
> maxlet = snd.maxpair.analyse
```

For example:

```
Main> maxlet "the quick brown fox jumped over the lazy sleeping dog"
'E'
(6265 reductions, 5626 cells)
```

`maxlet` could be used to help to decipher a ciphertext by supposing that the most frequent letter of the ciphertext corresponds to “E” in the plaintext. However, in general, more sophisticated methods are needed.

2.3 Other techniques

Despite needing to find just one letter, using only a frequency analysis would not guarantee deciphering a ciphertext. Generally, E is the most common letter in English words but it is possible to find pieces of text which have few occurrence of the letter E. So to completely decipher a plaintext, it is necessary to consider other ways of deciding what individual letters are.

A starting point for further analysis could be to create a list of potential plaintexts. This is done by finding the most frequent letter in the ciphertext (by using frequency analysis) and supposing that it was one of the most frequent letters in English. This would create the required list of plaintexts. These potential plaintexts could then be studied further. An alternative could be to suppose one of the six most frequent letters in the ciphertext corresponds to “E”. This would fail if the frequency of “E” in the plaintext is low.

Of course, the easiest way to see the plaintext is to generate all 26 possibilities - but this is against the spirit of this project and, for more complicated ciphers, it would be infeasible to output all the possible plaintexts.

2.3.1 Words

The implementation of the Caesar cipher means that any blanks in the original text are unchanged. This weakness can be exploited. Searches for one and two letter words and double letters within the text will help in analysing the text.

Using small words and double letters gives a way of assessing each of the potential plaintexts. Each word (or double letter) could be compared to a list of candidates and

each word that is not in the list would generate an error. The plaintext with the smallest number of errors is likely to be the correct ciphertext.

2.3.2 List of candidates

One letter words:

A I

Two letter words:

AM	AN	AS	AT	BE	BY	CO	DO	EG	GO	HE
HI	IE	IN	IS	IT	ME	MR	MY	NO	OF	OH
OK	ON	OP	OR	OX	PI	SO	TO	UP	US	WE

Common double letters:

CC	DD	EE	FF	GG	LL	NN	OO	PP	RR	SS	TT

2.3.3 Error checking

There needs to be a list of potential candidates for the tests. As the number of candidates is fairly small, they can be stored within the program itself:

```
> onelett =["A","I"]

> two1 =["AM","AN","AS","AT","BE","BY","CO","DO","EG","GO","HE","HI"]
> two2 = ["IE","IN","IS","IT","ME","MR","MY","NO","OF","OH","OK"]
> two3 = ["ON","OP","OR","OX","PI","SO","TO","UP","US","WE"]
> twolett = two1++two2++two3

> doubles = "CDEFGLNOPRST"
```

Also, the six most frequent letters in English needs to be listed:

```
> mostfreq :: [Int]
> mostfreq = map (\s -> ord s - ord 'A') ['E','T','A','I','N','O']
```

This gives the list `[4,19,0,8,13,14]`.

Next, a list of possible plaintexts need to be created using the six most frequent letters:

```
> posslist :: String -> [String]
> posslist ss = map (flip caesar ss) (map (+f) mostfreq)
>       where f = ord 'A'- ord(maxlet ss)
```

`flip` is a function defined in the Prelude, where:

```
flip        :: (b -> a -> c) -> a -> b -> c
flip f x y  = f y x
```

The Haskell prelude has a function words that chops up a string up into words (i.e. text separated by blanks).

To obtain words of size *n*, first define a boolean test that checks whether a string (list) is of size *n*.

```
> len :: Int -> [a] -> Bool
> len n ss = (length ss == n)
```

Now, use `len` and `words` to filter out words of length *n*.

```
> sizeword :: Int -> String -> [String]
> sizeword n ss = filter (len n) (words ss)
```

For example:

```
Main> sizeword 2 "to be or not to be"
["to","be","or","to","be"]
(770 reductions, 1148 cells)
```

To find all the double letters in a text, use:

```
> doub :: String -> String
> doub [] = []
> doub [x] = []
> doub (x:y:xs) = if (x==y) then x:doub xs else doub (y:xs)

Main> doub "bookkeeping"
"oke"
(89 reductions, 133 cells)
```

To work out the number of errors for each test, the appropriate list is created and compared with the list of candidates. The number of errors can be obtained by counting up the number of elements in the created list which do not match a candidate. To assist in this process, it is helpful to create a function which checks whether an element is not a member of a given list:

```
> notmember :: Eq a => [a] -> a -> Bool
> notmember [] _     = True
> notmember (x:xs) y = if (x == y) then False else notmember xs y
```

To find the numbers of errors in a given string:

```
> errors1 :: String -> Int
> errors1 ss = length (filter (notmember onelett) (sizeword 1 ss))

> errors2 :: String -> Int
> errors2 ss = length (filter (notmember twolett) (sizeword 2 ss))

> errors3 :: String -> Int
> errors3 ss = length (filter (notmember doubles) (doub ss))

> errors :: String -> Int
> errors ss = (errors1 ss) + (errors2 ss) + (errors3 ss)
```

The last thing to do is to create the number of errors for each potential plaintext and pick the text with the lowest number of errors. A list of pairs consisting of elements of the form (*Text*, *Errors in Text*) will be created by the mapping `err`:

```
> err :: String -> (String,Int)
> err a = (a, errors a)
```

Next, functions will be needed to find the element with the lowest second component:

```
> minsnd :: Ord f => (e,f) -> (e,f) -> (e,f)
> minsnd (a,b) (c,d) = if b <=d then (a,b) else (c,d)

> minpair :: [(a,Int)] -> (a,Int)
> minpair = foldr1 minsnd
```

(These functions are analogous to `maxfst` and `maxpair` defined in 2.2.1)

Finally, a definition for `decode`:

```
> decode :: String -> String
> decode = fst.minpair. (map err). (posslist)
```

To demonstrate the functions, the string `"LG TW GJ FGL LG TW LZSL AK LZW IMWKLAGF"` will be used:

Doing a frequency analysis:

```
Main> analyse "LG TW GJ FGL LG TW LZSL AK LZW IMWKLAGF"
[(2,'A'),(0,'B'),(0,'C'),(0,'D'),(0,'E'),(2,'F'),(5,'G'),(0,'H'),(1,'
I'),(1,'J'),(2,'K'),(7,'L'),(1,'M'),(0,'N'),(0,'O'),(0,'P'),(0,'Q'),(
0,'R'),(1,'S'),(2,'T'),(0,'U'),(0,'V'),(4,'W'),(0,'X'),(0,'Y'),(2,'Z'
)]
(4750 reductions, 5147 cells)
```

Finding the letter with the greatest frequency:

```
Main> maxlet "LG TW GJ FGL LG TW LZSL AK LZW IMWKLAGF"
'L'
(4159 reductions, 3848 cells)
```

Generating a list of possible texts:

```
Main> posslist "LG TW GJ FGL LG TW LZSL AK LZW IMWKLAGF"
["EZ MP ZC YZE EZ MP ESLE TD ESP BFPDETZY","TO BE OR NOT TO BE THAT
IS THE QUESTION","AV IL VY UVA AV IL AOHA PZ AOL XBLZAPVU","ID QT DG
CDI ID QT IWPI XH IWT FJTHIXDC","NI VY IL HIN NI VY NBUN CM NBY
KOYMNCIH","OJ WZ JM IJO OJ WZ OCVO DN OCZ LPZNODJI"]
(22377 reductions, 27604 cells)
```

(The second text looks the most likely!)

To look at the number of “errors” in each list:

```
Main> map errors (posslist "LG TW GJ FGL LG TW LZSL AK LZW IMWKLAGF")
[6,0,6,6,6,6]
(38185 reductions, 52016 cells)
```

The second text has the least number of errors.

Finally, cracking the text:

```
Main> decode "LG TW GJ FGL LG TW LZSL AK LZW IMWKLAGF"
"TO BE OR NOT TO BE THAT IS THE QUESTION"
(38907 reductions, 52923 cells)
```

2.4 Using Files

Cryptographic algorithms are often used on large pieces of text - checking the functions in the last section meant using the same piece of text repeatedly. It is useful to have the facility to read in a piece of text and then apply the cipher to the text. The output from an algorithm will be either a string of letters or a list of numbers. It is also beneficial to having a routine which will save the output from a cipher and then read it back in when decoding.

2.4.1 Files in Haskell

Functional file handling makes use of the IO() monad. It is necessary to convert between strings and IO().

Some useful functions:

```
> isText :: Char -> Bool
> isText c
>   | ord c < 32         = False
>   | ord c > 127        = False
>   | otherwise          = True
```

- This filters out any unwanted characters.

```
> file :: String -> String -> (String -> String) -> IO()
> file infn outfn func = do xs <- readFile infn
>                           writeFile outfn (filter isText (func xs))
```

- This is used to read in a file *infn*, apply the function *func* to the text of *infn* and then save the result in the file *outfn*.

```
> fileshow :: String -> (String -> String) -> IO ()
> fileshow infn func = do xs <- readFile infn
>                         putStr (filter isText (func xs))
```

- This reads in a file of name *infn*, applies the function *func* and then displays the result to the screen.

These 3 functions and the Prelude function

```
writeFile :: FilePath -> String -> IO ()
```

allow functions to be written which deals with files.

In this and subsequent sections, a standard convention will be used:

S represents an action to or from the screen

and **F** represents an action with a file.

So `caesarSF` means a function that takes an input from the screen and outputs it to a file. `caesar` stands for `caesarSS`.

Using this convention, six functions can be defined:

```
> caesarSF :: Int -> String -> String -> IO()
> caesarSF n ss outfn = writeFile outfn (show(caesar n ss))

> caesarFS :: Int -> String -> IO()
> caesarFS n infn = fileshow infn (show.(caesar n))

> caesarFF :: Int -> String -> String -> IO()
> caesarFF n infn outfn = file infn outfn (caesar n)

> decodeSF :: String -> String -> IO()
> decodeSF ss outfn = writeFile outfn (show(decode ss))

> decodeFF :: String -> String -> IO()
> decodeFF infn outfn = file infn outfn decode

> decodeFS :: String -> IO()
> decodeFS infn = fileshow infn decode
```

The file handling functions for the other implementations have a similar structure. The actual functions will not always be stated, however they can be found in the relevant place in the program listings - contained in Appendix A.

2.5 Testing the Caesar cipher programs

This section shows the output obtained from the functions developed in this chapter - particularly using the file handling routine. As is the case with functional programs, individual parts can be tested. The test files *file1*... *file4* are shown in Appendix D.

2.5.1 Using the file handling functions

a) Shifting *file4* by 15 letters (note that the letters are converted to uppercase)

```
Main> caesarFS 15 "file4"
"IWT FJXRZ QGDLC UDM YJBETS DKTG IWT APON HATTEXCV SDV"
(6169 reductions, 7599 cells)
```

b) Shifting *file3* by 23 letters saving the result to *file3cae*

```
Main> caesarFF 23 "file3" "file3cae"
(20399 reductions, 25079 cells)
```

c) Decoding *file3cae*

```
Main> decodeFS "file3cae"
THE DOCTOR PUTS HIS HAND ON ACES SHOULDER BEFORE THEY WENT INTO THE CHURCH
TIME TO LEAVE HE SAID ACE LOOKED INTO THE DOCTORS GREY EYES WE DID GOOD
DIDNT WE PERHAPS SAID THE DOCTOR TIME WILL TELL IT ALWAYS DOES
(179410 reductions, 239860 cells, 1 garbage collection)
```

[Notice that all the punctuation has been removed and the text converted to upper case letters.]

d) Doing a frequency analysis on *file2* (note no “E’s)

```
Main> fileshow "file2" (show.analyse)
[(24,'A'),(6,'B'),(12,'C'),(11,'D'),(0,'E'),(7,'F'),(7,'G'),(13,'H'),
(35,'I'),(1,'J'),(2,'K'),(21,'L'),(5,'M'),(19,'N'),(27,'O'),(9,'P'),(
1,'Q'),(12,'R'),(26,'S'),(23,'T'),(18,'U'),(3,'V'),(11,'W'),(0,'X'),(
7,'Y'),(0,'Z')]
(46540 reductions, 41950 cells)
```

e) Decoding *file2cae* (created by a shift of 7)

```
Main> decodeFS "file2cae"
INCURABLY INSOMNIAC ANTON VOWL TURNS ON A LIGHT WITH A LOUD AND LANGUOROUS
SIGH VOWL SITS UP STUFFS A PILLOW AT HIS BACK DRAWS HIS QUILT UP AROUND HIS
CHIN PICKS UP HIS WHODUNIT AND IDLY SCANS A PARAGRAPH OR TWO BUT JUDGING ITS
PLOT IMPOSSIBLY DIFFICULT TO FOLLOW IN HIS CONDITION ITS VOCABULARY TOO
WHIMSICALLY MULTISYLLABIC FOR COMFORT THROWS IT AWAY IN DISGUST
(312752 reductions, 417572 cells, 1 garbage collection)
```

Not all texts can be decoded. Decoding could fail if:

- The text contains few occurrences of the "most frequent" letters.
- There are no one or two letter words.
- The text does not consists of standard English.

2.5.2 An interactive Caesar function

The Haskell prelude contains a function:

```
interact :: (String -> String) -> IO ()
```

This can be used so that the Caesar cipher can be applied instantly to the keys that the user types in. A termination key will be required and '#' will be used.

So, an interactive version of the Caesar cipher can be specified:

```
> icaesar :: Int -> IO()
> icaesar n = interact ((caesar n).(takeWhile (/='#')))
```

The cipher will work on letters, leave spaces alone, terminate on '#' and ignore all other symbols.

For example,

```
Main> icaesar 15
PQR STUVW
(909 reductions, 1108 cells)
```

[The keys pressed were: `abc 3def*(gh#`]

2.6 Problems with the Caesar cipher

As can be seen from this chapter, the Caesar cipher is a very simple cipher to crack. The creation of a cracking program is not very difficult. The main techniques used to decode were using a frequency analysis and looking at one and two letter words. Any cipher which transposes the letters in the text can be broken by using frequency analysis. A better cipher would have to destroy the frequency of the letters of the original text or remove the blanks.

Chapter 3 The Vigenère Cipher

This chapter will look at a cipher – the Vigenère cipher – that is based on the principles of the Caesar cipher. A functional program to encipher a message using the traditional Vigenère cipher will be developed. The cracking will follow the methods of the first practical in the *Procedural Programming* course. However, instead of using Oberon, Haskell will be used. The methods outlined in this course use a modification of the Vigenère cipher.

3.1 The Fall of Caesar

As seen in the previous chapter, the Caesar cipher is very insecure. The main weakness with the Caesar cipher is that by performing a frequency analysis on the ciphertext, the most frequent letters can be determined. A more secure system needs to hide the frequency of the letters of the plaintext. One way to do this is to use different alphabets for differently placed letters in the plaintext. One cipher that does this is the *Vigenère cipher*.

3.1.1 Description of the Vigenère cipher

The Vigenère cipher [Sgh] uses a word as its key. Each letter of the key word acts as a shift for a Caesar cipher. The plaintext is written down with the key repeatedly written above the plaintext. Each letter of the plaintext is shifted according to what letter it is below (A represents a shift of 0, .., Z a shift of 25). To work properly, blanks need to be removed.

For example, if the plaintext is THE BODY IN THE LIBRARY and the key MARPLE, then the message is encrypted as follows:

Key	`MARPLEMARPLEMARPLEM`
Plaintext	`THEBODYINTHELIBRARY`
Ciphertext	`FHVQZHKIEISIXISGLVK`

The frequency of the letters in the plaintext cannot be determined. From the example, it can be seen that “I” in the ciphertext can represent “I”, “T” or “E” in the plaintext. This means that a standard frequency analysis outlined in Chapter 2 cannot be used here. In fact, the Vigenère cipher uses many different alphabets to encrypt its messages, hence it is an example of a *polyalphabetic* cipher (Caesar is a *monoalphabetic* cipher).

As the Vigenère cipher is based on the techniques used in the Caesar cipher, it should not be too difficult to construct algorithm which will encipher messages using the Vigenère cipher.

3.1.2 Functional Vigenère cipher

The program for the Vigenère cipher in Haskell will make use of the functions `capitalise`, `caps` and `nlett` from Section 2.1.2.

As letters are used instead of numbers, a function will be needed to convert a letter into a number - this is so that `nlett` can be used.

```
> toval :: Char -> Int
> toval c = ord c - ord 'A'
```

To define a Vigenère cipher, first use a function which uses `nlett` with the key and the plaintext:

```
> vig' :: String -> String  -> String
> vig' _ ""  = ""
> vig' (k:ks) (t:ts) = (nlett (toval k) t) : vig' ks ts
```

It is necessary to keep repeating the key when encrypting the plaintext. The Prelude has a function `cycle`:

```
cycle            :: [a] -> [a]
cycle []          = error "Prelude.cycle: empty list"
cycle xs          = xs' where xs'=xs++xs'
```

For example, `cycle "abcd" = "abcdabcdabc.......".`

Only upper case letters are encrypted - a filter is used to remove other characters. So, the function for the Vigenère cipher is:

```
> vig :: String -> String -> String
> vig ks ss = vig' (cycle(cap ks))  (filter isUpper (cap ss))
```

For example,

```
Main> vig "marple" "the body in the library"
"FHVQZHKIEISIXISGLVK"
(2037 reductions, 2620 cells)
```

3.1.3 Using the key to decrypt a Vigenère cipher

If the key is known, then a deciphering processing is quite straightforward. The only work required is to convert the key so that it can be used with the function `vig`.

For example, to decrypt the ciphertext above, the key needed is "OAJLPW".

The following table can be used to convert the encrypting key:

Encrypting	A	B	C	D	E	...
Decrypting	A	Z	Y	X	W	...

a function `revlet` can be obtained:

```
> revlet :: Char -> Char
> revlet 'A' = 'A'
> revlet c = chr (ord 'B' + ord 'Z' - ord c)
```

This can be used to specify a function which reverses a key:

```
> revkey :: String -> String
> revkey xs = map (revlet) (cap xs)
```

Now, a function can be defined for decrypting a Vigenère cipher:

```
> vigrev :: String -> String -> String
> vigrev k ss = vig (revkey k) ss
```

Using the example above,

```
Main> revkey "marple"
"OAJLPW"
(352 reductions, 433 cells)

Main> vigrev "marple" "FHVQZHKIEISIXISGLVK"
"THEBODYINTHELIBRARY"
(1759 reductions, 2350 cells)
```

A harder task, when decrypting a Vigenère cipher, would be to insert all the spaces that occur in the plaintext back into the correct places!

3.1.4 Weaknesses of the Vigenère cipher

Despite being more secure than the Caesar cipher, the Vigenère cipher still has weaknesses. Looking again at the example in Section 3.1.1:

Key	MARPLEMARPLEMARPLEM
Plaintext	THEBODYINTHELIBRARY
Ciphertext	FHVQZHKIEISIXISGLVK

In this “Y” twice is mapped to “K” and “I” is also mapped to “I” twice. The periods of repeat between these letters are 12 and 6 respectively, which are both multiples of the key length (i.e. 6). Finding the lengths of repeats in a ciphertext can give a list of candidates for the length of the key. Further analysis can then be performed to try to determine the key.

3.2 Cracking using Procedural Programming methods

The first practical of the *Procedural Programming* course [P1] was concerned with developing an encryption program in Oberon for a different form of the Vigenère cipher and the developing programs to crack the cipher.

3.2.1 A variation on the Vigenère cipher

The practical used a modified version of the Vigenère cipher. This cipher still uses a word as the key but it encrypts all the characters in a piece of text including the blanks.

The text is encrypted using ‘exclusive or’ (xor), denoted by $\oplus$. This function is used on binary digits (actually it is usually used with truth values; 1 can represent TRUE and 0 can represent FALSE) and has the rules:

$0 \oplus 0 = 0 = 1 \oplus 1$ and $0 \oplus 1 = 1 = 1 \oplus 0$

$\oplus$ is commutative and associative. It also obeys the law:

$(a \oplus b) \oplus b = a$ [the cancelling law],

which is useful in encryption - applying the same key twice gives back the original text.

⊕ can be used with decimal numbers by converting the numbers to binary and then using ⊕ bitwise and then converting back to decimal.

To develop functions which mimic ⊕, it is necessary to consider the binary representation of the integers used and checking the matching bits. This is equivalent to recursively checking the remainder mod 2, dividing the numbers by two and then rechecking. A list is made which consists of just 0s and 1s that can then be converted back to an integer.

The following functions implement ⊕:

```
> xor :: Int -> Int -> Int
> xor a b = twopw(xor' a b)

> xor' :: Int -> Int -> [Int]
> xor' 0 0 = []
> xor' a b = if mod a 2 == mod b 2 then 0:xx
>                                  else 1:xx
>                  where xx = xor' (div a 2) (div b 2)

> twopw :: [Int] -> Int
> twopw = foldr (\ a b -> a + (2*b)) 0
```

To use ⊕ with characters:

```
> chars :: Char -> Char -> Char
> chars a b = chr (xor (ord a) (ord b) )
```

If both *m* and *n* are 7 bit (i.e. between 0 and 127) then so will be the result.
To create a cipher:

ciphertext = plaintext ⊕ key'
where *key'* is the repeated key.

⊕ is used character-wise on each element of the plaintext.

3.2.2 Creating a ciphertext.

The function `chars` can be used to turn a plaintext into a ciphertext using a key (in fact a ciphertext can be turned back into the corresponding plaintext by using the same functions and the same key).

```
> vig :: String -> String -> String
> vig ks ss = vig' (cycle ks) ss

> vig' :: String -> String -> String
> vig' _ []          = []
> vig' (t:ts) (s:ss) = (chars t s) : (vig' ts ss)
```

The usual output is just gibberish, so file handling versions are used. Slight changes are needed from the functions in Section 2.4.1, as non-text characters cannot be

filtered out - the functions can be found in Appendix A.

3.2.3 Using a crib

One method the *Procedural Programming* practical applied to crack this Vigenère cipher was the use of a **crib**. The idea is to use a suspected piece of the plaintext (called a crib) to try and recover the key. This feels like little bit like cheating! However, during the Second World War, German ciphers were often broken by guessing what certain pieces of the ciphertext were. In particular, the first message of the day that was sent was usually the weather report. By supposing that the plaintext contained the German word "wetter", information could often be gained about the text. As the key for the ciphers were not normally changed throughout the day, breaking the first message of the day gave a tremendous advantage [Sgh].

Using the example in Section 3.1.1 (looking simply at letters), the ciphertext was:

```
FHVQZHKIEISIXISGLVK
```

Suppose that it is known that "BODYINTHE" occurs in the plaintext. This can then be used to try to find the key by trying this piece of text at various positions of the ciphertext. The aim of this checking is to find a repeating pattern for the key - to do this the length of the crib must be greater than the key length. [In this example, the reversing can be done by using a function such as `vigrev,` defined in Section 3.1.2. For the Crypt cipher, ⊕ can be used with the cancelling law].

Suppose that the piece was tried starting at the second place in the ciphertext:

```
Piece of text      BODYINTHE
Ciphertext       FHVQZHKIEISIXISGLVK
Implied key        GHNBZXPXE
```

No repetition occurs here, so this not likely to be the correct place.

However, if checking was started at the fourth place in the ciphertext:

```
Piece of text         BODYINTHE
Ciphertext       FHVQZHKIEISIXISGLVK
Implied key           PLEMARPLE
```

Here a clear repeat can be seen and the period of repeat corresponds to the length of the key. The key can be determined by working back along the ciphertext.

The first function needed checks for a repeat. To help to eliminate accidental matching, the function implemented will only check for a repeat of at least two characters. The period of repetition is needed as this corresponds to the length of the key - this is worked out by the function `replen` (0 denotes no repeat).

```
> rep :: Eq a => [a] -> Int -> Bool
> rep xs n = and (zipWith (==) xs (cycle (take n xs)))
```

```
> test :: Int -> [Bool] -> Int
> test _ []    = 0
> test n (x:xs)= if x == True then n else (test (n+1) xs)

> replen :: Eq a => [a] -> Int
> replen xs = test 1 (map (rep xs) [1..(length xs)-2])
```

For example:

```
Main> replen "abcabcab"
3
(353 reductions, 519 cells)
Main> replen "abcabcac"
0
(838 reductions, 1267 cells)
Main> replen "abcda"
0
(326 reductions, 491 cells)
```

Now, the crib needs to be tried at each possible position of the ciphertext. The resulting text needs to be checked to see whether it repeats. The next function makes a list of triples consisting of (*key*, *length of repeat, position*).

```
> criblist :: Int -> Int -> String -> String -> [(String,Int,Int)]
> criblist 0 _ _ _       = []
> criblist 0 _ _ []      = []
> criblist k n cs (s:ss) = (vig ss cs, replen (vig ss cs), n-k):
>                              criblist (k-1) n cs ss
```

The syntax for `criblist` is:

```
      criblist (counter) (max. position) (crib) (text)
```

file1 has been encrypted with a key and the result has been stored in *file1code* - this will be used as an example.

The crib “chance led” will be used with *file1code* - the first 10 positions are shown.

```
Main> fileshow "file1code" (show.(criblist 10 10 "chance led"))
[("wip'nj2Jgr",0,0),("by(clw\ACKns0",0,1),("r!laqC\"z1a",0,2),("*en|E
g68`o",0,3),("ngsHasting",8,4),("lzGlu1%gf7",0,5),("qNcx7`+o6A",0,6),
("Ejw:fn#?@|",0,7),("a~5khfsI}7",0,8),("u<de`6\ENQt6v",0,9)]
(68034 reductions, 91252 cells)
```

It is now necessary to choose the correct element of the list created by the function `criblist`. The element wanted is the one with a non-zero second component. If no such element exists then ‘-1’ is returned.

```
> head1:: [(String,Int,Int)] -> (String,Int,Int)
> head1 [] = ("",-1,-1)
> head1 (x:xs) = x

> snd' :: (a,Int,c) -> Bool
> snd' (a,b,c) = b==0

> crib' :: String -> String -> (String,Int,Int)
> crib' cs ss = head1(dropWhile snd' (criblist k k cs ss))
>              where k = length ss - length cs
```

Using *file1code*:

```
Main> fileshow "file1code" (show.(crib' "chance led"))
("ngsHasting",8,4)
(27677 reductions, 36371 cells)
```

The element returned from `crib'` is of the form (*string*, *key length*, *position in text*) (or (“”,-1,-1) if a key cannot be found).

The key can now be obtained from the first component of `crib'`.

```
> crib :: String -> String -> String
> crib cs ss = if k>0 then take k (drop (k-(mod (p+1) k))
>                                       (cycle (take k xs)))
>               else []
>               where (xs,k,p) = crib' cs ss
```

To find the key used on *file1:*

```
Main> cribFS "chance led" "file1code"
"Hastings"
(27456 reductions, 36107 cells)
```

Checking this:

```
Main> vigFS "Hastings" "file1code"
Pure chance led my friend
......
Victory Ball.
(179044 reductions, 230907 cells)
```

It is possible for the `crib` function to fail. The crib could be too short or it could give an incorrect key if the crib contains repeated characters.

```
Main> cribFS "chance le" "file1code"
""


Main> cribFS "widespread" "file1code"
"U1vx|bd\DEL"

Main> cribFS "widespread " "file1code"
"Hastings"
```

3.2.4 Finding the length of the key

The aim of this section is to try to find the length of the key by shifting the ciphertext by a set amount and counting how many matches occur. (Remember in Section 3.1.4, the matches occurred with shifts of multiples of the key length). The shift with the greatest number of matches is likely to correspond to the key length (or a multiple of the key length).

The first step is to define a function which counts the number of matches between two strings. A “match” is counted if the strings have the same character in the same place.

```
> matches :: String -> String -> Int
> matches [] _          = 0
> matches _ []          = 0
> matches (s:ss) (t:ts) = if s==t then 1 + xx else xx
>                           where xx = matches ss ts
```

The text needs to be shifted one place at a time and the number of matches needs to be counted.

```
> shifts :: String -> String -> [Int]
> shifts [] _  = []
> shifts ss ts = matches ss ts : shifts (tail ss) ts
```

Simply defining a function such as:

```
      analyse ss = shifts (tail ss) ss
```

would just out a (potentially long) list of numbers.

A better idea is to define a function that allows the list to be displayed with a label indicating the position in the string:

```
> display :: Show a =>[a] ->  Int -> Int -> String
> display  _ _ 0     = []
> display [] _ _     = []
> display (t:ts) n e =
>         ((show n)++":\t"++(show t)++"\n")++display ts (n+1)(e-1)
```

'\t' is the tab character and '\n' represents newline.

The syntax for this function is:

```
       display (string) (start label) (no. of elements)
```

For any examples used, the first 25 shifts will be considered.

Finally:

```
> analyse' :: String -> String
> analyse' ss = display (shifts (tail ss) ss) 1 25

> analyse :: String -> IO()
> analyse = putStr.analyse'
```

For an example, *file3* has been encrypted with a key word and the result is stored in *file3code*.

```
Main> analyseFS "file3code"
1:      2
2:      6
3:      3
4:      22
5:      3
6:      3
7:      5
8:      15
```

```
............
22:      0
23:      4
24:      12
25:      0
(18220 reductions, 30157 cells)
```

From the results above, it seems that the length of the key is 4 and this will be used in the next section.

3.2.5 Finding the key

Once a potential key length is known then it is necessary to try to guess the characters that occur in the key word. Again, the ciphertext is shifted (by multiples of the key length) and matches are looked for. By exploiting the fact that blanks are frequent in written English, characters are guessed by supposing that they correspond to a blank in the plaintext (if considering blanks is inconclusive then "E" could be searched for - although both upper and lower case "E"s would have to be used). Only printable characters (i.e. those with ASCII values between 32 and 127) are considered.

The first function test for equality. If the strings match then the place where they match is recorded as the second component of a tuple.

```
> eq :: Int -> String -> String -> [(Char,Int)]
> eq _ [] _           = []
> eq _ _ []           = []
> eq n (s:ss) (t:ts) = if (s == t) then (s,n):xx else xx
>                            where xx = eq (n+1) ss ts
```

The next function is used with map. On the first component, it tries to work out what the character in the key would be; the second component is taken modulo *k* (the key length).

```
> tries :: Int -> (Char,Int) -> (Char,Int)
> tries k (a,b) = (chars a ' ',mod b k)
```

Now the list of possible characters in the keyword needs to be worked out:
[`isText` was defined in Section 2.4.1.]

```
> list' :: Int -> String -> String -> [(Char,Int)]
> list' _ [] _  = []
> list' k ss ts = (filter (isText.fst)(map(tries k)(eq 0 ss ts))) ++
>                          (list' k (drop k ss) ts)

> list :: Int -> String -> [(Char,Int)]
> list k ss = list' k (drop k ss) ss
```

For example,

```
Main> listFS 4 "file3code"
[('<',2),('!',3),('!',2),.......
```

It is quite hard to pick a potential keyword from the list, so it might to useful to know how many times each pair occurs. A third component could be added to each element in the list which corresponds to the frequency.

```
> accum :: [(Char,Int)] -> [(Char,Int,Int)]
> accum [] = []
> accum ((a,b) :ss) = (a,b,c) : accum ss'
>                   where c = 1+(length(filter(==(a,b)) ss))
>                         ss' =  filter (/= (a, b)) ss
```

- The list obtained from this function has elements of the form (*Character*, *Position*, *Frequency*).

It is useful to filter out those elements which do not occur very often (i.e. those with a low third component):

```
> best :: Int -> Int -> String -> [(Char,Int,Int)]
> best n k ss = filter3rd n (accum (list k ss))
```

where

```
> gt3rd :: Ord c => c -> (a,b,c) -> Bool
> gt3rd n (a,b,c) = c>=n

> filter3rd :: Int -> [(Char, Int,Int)] -> [(Char,Int,Int)]
> filter3rd n [] = []
> filter3rd n ss = filter (gt3rd n) ss
```

The syntax is

```
      best (min no of occurrences) (key length) (text)
```

Using the same example:

```
Main> bestFS 10 4 "file3code"
[('<',2,10),('!',3,66),(';',2,10),('W',0,21),('o',2,105),('h',1,15),('d',3,10),('\'',1,15),('-',1,15),('*',2,15),('u',3,10)]
(222398 reductions, 300033 cells, 1 garbage collection)
```

The key is likely to be the characters with the greatest third component (the frequency) for each place in the text (corresponding to the second component). To define a function which tries to find the key, first use:

```
> maxs :: Ord c => (a,b,c) -> (a,b,c) -> (a,b,c)
> maxs (a,b,c) (d,e,f) = if c <f then (d,e,f) else (a,b,c)

> maxlist ::  [(Char,Int,Int)] -> (Char,Int,Int)
> maxlist = foldr (maxs) ('?',0,0)

> first :: (a,b,c) -> a
> first (a,b,c) = a

> third :: (a,b,c) -> c
> third (a,b,c) = c

> eq2nd :: Eq b => b -> (a,b,c) -> Bool
> eq2nd n (a,b,c) = n == b
```

The function below returns the character which has the largest third component for those elements which have a second component equal to *n*. If there are no elements which have a second component equal to *n*, then '?' is returned.

```
> find :: Int -> [(Char,Int,Int)] -> Char
> find n ss = (first (maxlist(filter (eq2nd n) ss)))
```

Finally, a function which returns a likely key:

```
> key :: Int -> String -> String
> key k ss = reverse(key' k  (best 1 k ss))

> key' :: Int -> [(Char,Int,Int)] -> String
> key' 0 _  = []
> key' n ss = find (n-1) ss: (key' (n-1) ss)
```

Using *file3code,*

```
Main> keyFS 4 "file3code"
"Who!"
(223179 reductions, 300512 cells, 1 garbage collection)
```

Checking this key:

```
Main> vigFS "Who!" "file3code"
The Doctor
......
always does."
(60540 reductions, 78101 cells)
```

This gives the correct answer.

It is possible that this function will fail. Failure could occur if the plaintext does not contain many spaces or if the list generated by `best` contains elements which have a large third component that were not in the original keyword.

3.3 Problems with the Vigenère cipher

Even though the Vigenère cipher is more secure than the Caesar cipher, it is still vulnerable to attack. The cipher still keeps the characteristics of the plaintext (such as the frequency of blanks) which makes it susceptible to attack. Once the key has been found, it is very simple to decrypt the ciphertext.

Chapter 4 From Private To Public

In the previous chapters, two algorithms were discussed. The purpose of this chapter is to take a brief look at some other algorithms (without implementing them in Haskell) and to discuss why a new type of cryptographic system is needed.

4.1 Other Private Key Algorithms

The Caesar cipher and the Vigenère cipher are both examples of **private key** algorithms. A *private key* algorithm is one in which the encryption and decryption of a text uses the same key. Hence, the key must be kept secret and only given to the people who need it. This section briefly looks some other private key algorithms.

4.1.1 Substitution Cipher

To use a substitution cipher [Wsh], it is necessary to form a permutation of the alphabet to use as the key.

For example, if the permutation was:

W U D O A H L M ...

then in the message, A would be substituted by W,..., F would be substituted by H and so on.

As with the Caesar cipher, this substitution cipher is monoalphabetic and so it is vulnerable to frequency analysis. It also can be quite difficult to remember the order of the letters which constitute the key.

To overcome the vulnerability, it is possible to produce adaptations of this cipher that are harder to crack.

4.1.2 The Playfair Cipher

One such adaptation of the substitution cipher replaces digrams (groups of two letters) instead of single letters. This type of substitution masks the frequency of the individual letters and so makes decryption harder. One such example is the Playfair cipher [Wsh]. To use this, needs a grid of letters. I and J are usually put in the same cell in the grid.

```
B Q L A I/J
W P Z C R
S D O K N
E H M T X
G U F Y V
```

To encrypt each digram, it is necessary to find the position of each of the letters in the grid. There are a number of ways to form the ciphers. Here is one way:

- If the letters form the diagonal of a rectangle, then the opposite corners form the code - e.g. CM is mapped to ZT.
- If the letters are in the same row then the letters to the immediate right are taken (wrapping around where necessary). Similarly, if the letters occur in the same column then the letters below are taken - e.g. RP is mapped to WZ .
- If a letter is repeated then another letter (such as X) can be inserted between the repeats.

For example:

To encrypt THE LITTLE GREY CELLS
break the text into digrams
TH-EL-IT-XT-LE-GR-EY-CE-LX-LS
and then use the grid
XM-MB-AX-EX-BM-VW-TG-WT-IM-BO

So, the encrypted message would be XMMBAXEXBMVWTGWTIMBO.

To decrypt, the receiver needs a copy of the grid and simply reverses the process.

Unfortunately, this cipher is also breakable. The cipher is cracked by looking at common digrams in English.

4.1.3 The Book cipher

The Book cipher [Sgh] uses (not surprisingly) a book to generate a sequence of letters. One way to use the cipher is to label each word in a section of a book with a number. Each number corresponds to the letter which starts the word. There could be various numbers for a particular letter and so a frequency analysis would not work on the text.

The key in this kind of cipher consists of the book used and the starting place in the book. If the sequence of letters does not contain all the letters of the alphabet then it may not be possible to encipher the plaintext.

4.1.4 The One-Time Pad

The One-Time Pad is quite a remarkable cipher. If used properly, it is unbreakable [Wsh]. To use the One-Time Pad, it is necessary to generate a sequence of random letters (i.e. each letter of the alphabet has an equal chance (=1/26) of occurring). Each letter of the plaintext is shifted according to the corresponding letter in the random sequence (in the same way as the key is used in the Vigenère cipher).

For example,
To encrypt the message
BE SEEING YOU
using the sequence
AIENDSKLZW

Plaintext	B	E	S	E	E	I	N	G	Y	O	U
Sequence	A	I	E	N	D	S	K	L	A	Z	W
Ciphertext:	B	M	W	R	H	A	X	R	Y	N	Q

To ensure complete security, a different One-Time Pad must be used for each message that is sent.

Despite being completely secure, the One-Time Pad has a number of drawbacks:

- The sequence of random letters needs to be as long as the message

- There is no easy way of generating completely random letters - a pseudorandom routine may lose security.
- The sequence of letters must be distributed to everyone who needs to decode the message

4.2 The Key Distribution Problem

The main problem with the cryptographic methods mentioned so far is their reliance on the secrecy of the key. Once the key is known, the encrypted message can be deciphered very easily (assuming knowledge of the particular method is known). This means that the key has be distributed securely amongst the people who need to use it and this is the major drawback in private key algorithms – this is known as the **Key Distribution Problem** [Wsh, Sgh]. It would be better if the security of an encrypted message did not depend on knowledge of the key used to encrypt a message. This leads us to the creation of **public key algorithms**.

4.2.1 Public Key Algorithms

A *public key* algorithm usually has two keys. One key is used for encryption and the other for decryption. Each user of a system has a pair of keys. The encryption key (also known as the **public key**) can be made available to everyone (hence the term public) and the decryption key (also known as the **private key**) should only be known to the user.

The essential part of a public key system is the idea of a *one way* encryption function. The function needs to be easy to apply using the encryption key but it should be hard to undo the encryption even with knowledge of the encryption key. The function also needs to have a *trapdoor* - i.e. a way to invert the function using a decryption key.

The following chapters will look at different public key algorithms and how they may be implemented in Haskell.

Chapter 5 The Knapsack Algorithm

In the last chapter, the idea of a *public key system* was introduced. This chapter outlines one of the earliest public key systems and gives an implementation in Haskell.

5.1 The Knapsack Problem

This section looks at the development of a public key system based on the Knapsack problem [Wsh, Sch]. The **Knapsack Problem** can be stated in the following way:

Given positive integers S and $a_1, a_2, \ldots, a_n$ (called **weights**),
is is possible to find $x_1, x_2, \ldots, x_n$, where $x_i \in \{0,1\} \forall i$, such that $S = \sum_i (x_i \cdot a_i)$?

An alternative formulation is called the **Subset Sum**:
Given S and $A = \{a_1, \ldots, a_n\}$, is there a subset $X \subseteq A$ such that $S = \sum_{x_i \in X} x_i$?

This is considered to be a "hard" problem. How could this be used as the basis for a public key system?

5.1.1 Creating a system

One way to create a cryptographic system using the Knapsack problem could be as follows [Wsh]:

- The message first needs to be converted to binary. This could be achieved by using the ASCII values for each character in the message and then changing the values to 7 bits.

e.g. For ABC,
The ASCII values of the letters are 65, 66 and 67
The corresponding binary values are 1000001, 1000010 and 1000011
So ABC becomes 100000110000101000011.

- Now, choose a sequence of weights. If there are *n* weights, then the binary message needs to be split up into blocks of size *n*. A sum, like the one above, can be formed by making the bits in each block act as the *x* values. So, the message has been converted to a list of integers.

Unfortunately, it may not be possible to reverse this process and completely determine the original message even with the knowledge of the sequence of weights.

Suppose that the sequence of weights is [200, 10, 12, 23, 35, 45, 80] and the ciphertext contains the number 280.

280 could correspond to any or 1111100, 1000110, 1011010 or 1000001 (the characters 'l', 'E', 'Z' and 'A').

This is not acceptable for a cryptographic system.

5.1.2 Superincreasing sequences

A cryptographic system that uses the Knapsack problem needs to be easily reversible without giving any ambiguities in the plaintext.

One way to adapt the Knapsack problem is by using a *superincreasing sequence*.

A sequence $a_1, a_2, \ldots, a_n$ is said to be **superincreasing** if $\forall t\,(1 < t \leq n),\ a_t > \sum_{i=1}^{t-1} a_i$

The advantage of (a_i) being a superincreasing sequence is that there is a simple algorithm for finding the subset $X \subseteq A$ where $A = \{a_i\}$ and $S = \sum_{z_i \in X} a_i x_i$.

The algorithm goes as follows:

There are two cases to consider at each step -

(a) if $S \geq a_n$, then as $a_n > \sum_{i=1}^{n-1} a_i$, $a_n \in X$. Then look at $S - a_n$ and a_{n-1}.

(b) otherwise, $a_n \notin X$. Now look at S and a_{n-1}

Also, the set X is uniquely determined and so no ambiguities will arise.

This means that with a superincreasing sequence as a key, a message can be converted into a sequence of integers. Then, by using the algorithm outlined above, the ciphertext can be converted back into the message. However, this just leads to another private key system - as both encryption and decryption use the same key. A public key system using the Knapsack problem needs to have a superincreasing sequence as the decryption key but have a non-superincreasing sequence as the encryption key.

5.1.3 Making a public key system

To use the Knapsack problem as a public key system, it is necessary to define an encryption key and a decryption key.

The decryption key is a superincreasing sequence. To make an encryption key, the superincreasing sequence is multiplied by a number w modulo n.

The encryption process

- A superincreasing sequence, of length k, is transformed by multiplying by w modulo n.
- The plaintext text is converted into binary and split into blocks of size k.
- The modified sequence is used to convert each block into a number.
- The ciphertext is therefore a sequence of numbers.

The decryption process

- Each number in the ciphertext is multiplied by a number a with the property that $a \times w = 1 \pmod{n}$. So in fact $a = w^{-1}$ modulo n.
- The numbers are now converted to binary by using the algorithm outlined in Section 5.1.2.

- The binary numbers are converted back to characters.

Choice of multiplier and modulus

- n needs to be greater than the sum of the numbers in the superincreasing sequence, so that the modified sequence can be converted back without any ambiguity.
- w and n need to be coprime. This is so that w^{-1} exists modulo n. (By the Euclidean algorithm [Dbn], there are a and b such that $a\ w + b\ n = \gcd(w, n)$. So if w and n are coprime then $\gcd(w, n) = 1$ and $a = w^{-1} \pmod{n}$.)

5.2 Implementing the Knapsack algorithm in Haskell

This section will look at how to implement the Knapsack algorithm in Haskell. Two main functions will be developed - one to encipher a message using a sequence and one to decipher a ciphertext using a superincreasing sequence.

5.2.1 Creating superincreasing sequences

It is useful to have a function which will test whether a given sequence is superincreasing or not.

For a given sequence, it is necessary to find the partial sum at each stage and then compare to the next term.

`scanl` is a Prelude function that applies a function to the initial segments of a list

```
scanl            :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs      = q : (case xs of
                    []   -> []
                    x:xs -> scanl f (f q x) xs)
```

In particular, `scanl (+) 0` will calculate the partial sums of a sequence.

```
Main> scanl (+) 0 [1,2,3,4]
[0,1,3,6,10]
(81 reductions, 181 cells)
```

The following functions tests whether a given sequence is superincreasing or not:

```
> testSup :: Integral a=>[a]-> Bool
> testSup xs = and(zipWith (>) xs (scanl (+) 0 xs))

Main> testSup [1,4,8,20]
True
(90 reductions, 183 cells)
Main> testSup [1,2,4,7]
False
(83 reductions, 133 cells)
```

It is also useful to have a function, which will take in a sequence and make a superincreasing sequence:

```
> super :: Num a => [a] -> [a]
> super [] = []
> super (x:xs) = sup2 xs [x]

> sup2 :: Num a => [a] -> [a] -> [a]
> sup2 [] y = y
> sup2 (x:xs) y = sup2 xs (y++[1+x+(sum y)])
```

The first term of the created superincreasing sequence is the same as the original list. The remaining terms of the list are used to generate the rest of the sequence.

```
Main> super [1,0,0,0]
[1,2,4,8]
(116 reductions, 190 cells)
Main> super [1,5,3,0,1]
[1,7,12,21,43]
(155 reductions, 275 cells)
```

5.2.2 Encrypting

This section is concerned with implementing the encryption process which was outlined in Section 5.1.3. The first function needed converts an integer to a string representing the number in binary:

```
> bin' :: Int -> String -> String
> bin' 0 ys = ys
> bin' n ys = bin' q (chr(ord '0' + p) :ys)
>                where (q,p) = quotRem n 2

> bin :: Int -> String
> bin n = bin' n ""

Main> bin 29
"11101"
(198 reductions, 283 cells)
Main> bin 149
"10010101"
(303 reductions, 436 cells)
```

When creating the code, it is necessary to ensure that the size of the binary strings is the same for each character. Here is a function which packs in zeros at the front of the binary string, where necessary:

```
> zeros' :: Int -> String -> String
> zeros' 0 s = s
> zeros' n s = zeros' (n-1) ('0':s)

> zeros :: Int -> String -> String
> zeros n s = if n>l then zeros' (n-l) s else s
>            where l = length s

Main> zeros 7 (bin 29)
"0011101"
(310 reductions, 422 cells)
Main> zeros 10 (bin 149)
"0010010101"
(436 reductions, 599 cells)
```

```
Main> zeros 3 (bin 149)
"10010101"
(369 reductions, 509 cells)
```

In this implementation, the size of each binary string will need to be 7 (corresponding to a number between 0 and 127). [Note: if exactly 7 numbers constitute the encryption key sequence, then a frequency analysis can be used on the ciphertext as each number in the ciphertext will usually correspond to exactly one character in the plaintext.]

Now, a function which converts a text string into a binary string:

```
> input :: String -> String
> input "" =""
> input (x:xs) = (zeros 7(bin (ord x)))++input xs

Main> input "abc"
"110000111000101100011"
(980 reductions, 1405 cells)
```

The function `combine` takes a binary string and a sequence of numbers as its input. It produces an integer by matching up each place of the string with the numbers in the sequence and then summing the resulting list.

```
> combine :: String -> [Integer] -> Integer
> combine "" _          = 0
> combine _ []          = 0
> combine (s:ss) (x:xs) = (kn s x) + (combine ss xs)

> kn :: Num a => Char -> a -> a
> kn '1' n = n
> kn '0' n = 0

Main> combine "1011" [5,8,17,34]
56
(35 reductions, 54 cells)
```

$(56 = 5 + 17 + 34)$

When using the Knapsack algorithm, the long string of 1s and 0s needs to be split into blocks of size equal to the sequence length. The function `coding` takes a value *n*, a binary string and a sequence and chops the string into blocks of size *n* which are then put into `combine`. It is more efficient to have *n* defined as an input rather than using a `where` clause and defining `n=length xs` at each stage.

```
> coding :: Int -> String -> [Integer] -> [Integer]
> coding n "" xs = []
> coding n ss xs = (combine ys xs):(coding n zs xs)
>                  where (ys,zs)= splitAt n ss
```

The Prelude function `splitAt` is given by the rule:

```
splitAt n xs = (take n xs, drop n xs)
```

Finally, a function can be defined which takes in a text string and a sequence. The text string needs to be changed into a binary string, by use of the function `input`.

```
> knapsack :: [Integer] -> String -> [Integer]
> knapsack xs ss = coding n (input ss) xs
>                  where n = length xs

Main> knapsack [10,23,76,4] "Sherlock"
[86,103,86,80,76,113,76,37,14,90,113,4,109,90]
(3357 reductions, 5347 cells)
Main> knapsack [10,23,76,4,47] "Sherlock"
[86,113,57,14,113,74,99,84,113,127,133,10]
(3324 reductions, 5289 cells)
```

The function `knapsack` will work for any sequence of integers, but to enable a ciphertext to be decoded, a modified superincreasing sequence must be used.

It is useful to have a function which will convert a superincreasing sequence given a suitable multiplier and modulus - the function `convert` does this:

```
> convert :: Integral a=>[a] -> a -> a -> [a]
> convert xs w n
>   | gcd w n /= 1    = error "not coprime"
>   | n <= sum xs     = error "modulus too small"
>   | otherwise       = map (\a -> rem (a*w) n) xs
```

[A third test could be included using the function `testSup` to check whether `xs` is a superincreasing sequence].

Syntax: `convert (superinc seq) (multiplier) (modulus)`

```
Main> convert [4,9,19,35,79] 23 151
[92,56,135,50,5]
(314 reductions, 532 cells)
```

For completeness, here is a function will takes in a piece of text, a superincreasing sequence, a multiplier and a modulus as the input and applies the Knapsack coding after first converting the sequence into a non-superincreasing sequence:

```
> knapmod :: [Integer] -> Integer -> Integer -> String ->[Integer]
> knapmod xs w n ss= knapsack (convert xs w n) ss
```

Syntax: `knapmod (superinc seq)(multiplier)(modulus)(text)`

```
Main> knapmod [4,9,19,35,79] 23 151 "Dr. Watson"
[97,190,135,282,135,50,282,148,246,61,153,288,338,241]
(4362 reductions, 7034 cells)
Main> knapsack [92,56,135,50,5] "Dr. Watson"
[97,190,135,282,135,50,282,148,246,61,153,288,338,241]
(4118 reductions, 6544 cells)
```

5.3 Decrypting a Knapsack algorithm

The aim of this section is to produce Haskell functions that will, given the decryption key, decrypt a ciphertext created from the Knapsack algorithm.

5.3.1 The Euclidean Algorithm

So, for the Knapsack algorithm, the *decryption key* consists of a sequence (e_i), a multiplier w and a modulus n and the *encryption key* just comprises the sequence (d_i) where $d_i = w \times e_i \pmod{n}$. The first part of the decryption process is to multiply the each number in the ciphertext by the inverse of the multiplier. Since, the multiplier and the modulus are coprime, the inverse exists [Section 5.1.3]. To find the inverse, the Extended Euclidean algorithm [Dbn] will be used.

First, a description of the **Euclidean Algorithm**:

To find the greatest common divisor (gcd) of a and b :
(a) define $r_0 = b$
(b) find the integers q_1 and r_1 such that $a = q_1 b + r_1$
(c) now find q_2 and r_2 such that $(r_0 =) b = q_2 r_1 + r_2$
(d) continue the process until $\exists k$ s.t. $r_k = 0 \wedge r_{k-1} \neq 0$
(e) the gcd of a and b is r_{k-1}

The Haskell definition of `gcd` is along the following lines:

```
> gcd x y = if y==0 then x else gcd y (rem x y)
```

If the steps for the Euclidean algorithm are traced backwards, it is possible to express the gcd as a linear combinations of a and b - this is known as the Extended Euclidean algorithm. This finds $a^{-1} \pmod{b}$ if a and b are coprime (see Section 5.1.3).

(a) Suppose that r_{k-1} is the gcd of a and b.
Then $r_{k-3} = q_{k-1} r_{k-2} + r_{k-1}$ and so $r_{k-1} = r_{k-3} - q_{k-1} r_{k-2}$
(b) Looking at the previous step: $r_{k-4} = q_{k-2} r_{k-3} + r_{k-2}$
Then $r_{k-2} = r_{k-4} - q_{k-2} r_{k-3}$ and so $r_{k-1} = r_{k-3} - q_{k-1}(r_{k-4} - q_{k-2} r_{k-3})$
(c) Continuing this process enables the gcd to be expressed in terms of a and b.

To construct a function to work out the coefficient of a (the coefficient of b is not needed) first look at the forward and backwards steps together:

$$a = q_1 b + r_1 \qquad r_1 = a - q_1 b$$
$$b = q_2 r_1 + r_2 \qquad r_2 = b - q_2 r_1 = b - q_2(a - bq_1) = -aq_2 + b(1 + q_1 q_2)$$
$$r_1 = q_3 r_2 + r_3 \qquad r_3 = r_1 - q_3 r_2 = (a - q_1 b) - q_3(-aq_2 + b(1 + q_1 q_2))$$

At the kth stage, the new coefficient of a is worked by $a'' - (q \times a')$, where q is the quotient, a' the coefficient of a at stage k–1 and a'' the coefficient at stage k–2.

This leads to the function `euc`:

```
> euc' :: Integral a => a -> a -> a -> a -> a
> euc' b r c c1
>   | r == 0    = c1
>   | otherwise = euc' r p (c1-(q*c)) c
>                 where (q,p) = quotRem b r
```

```
> euc :: Integral a => a -> a -> a
> euc a b = mod (euc' a b 0 1) b
```

So, for w and n, where w and n are coprime (i.e. gcd (w, n) = 1),
$w \times (\text{euc } w\ n) = 1 \pmod{n}$

```
Main> euc 10 21
19
(151 reductions, 261 cells)
Main> mod (10*(euc 10 21)) 21
1
(218 reductions, 347 cells)
```

So, to change back a sequence which has been multiplied by a number, it is necessary to use `euc` in a function similar to `convert` (Section 5.2.2).

```
> unconvert :: Integral a=>[a] -> a -> a -> [a]
> unconvert  xs w n = map (\a -> rem (a*(euc w n)) n) xs

Main> convert [5,8,15,34] 13 70
[65,34,55,22]
(288 reductions, 483 cells)
Main> unconvert [65,34,55,22] 13 70
[5,8,15,34]
(902 reductions, 1630 cells)
```

5.3.2 Reconstructing the text

Once the superincreasing sequence has been recovered (by using `convert`), it is necessary to reconstruct the binary string, chop the string up into blocks of length 7, change each block back to a decimal ASCII value and so find the corresponding character.

Reconstructing the binary string can be achieved by using the algorithm outlined in Section 5.1.2 - remembering that '1' corresponds to a number in the sequence being included.

The first operation needed is one to construct the binary string. `makebin` takes an integer and a (superincreasing) sequence and creates a binary string using the algorithm outlined in Section 5.1.2. The intermediate stages in the fold produce elements of the form (*string*, *number*).

```
> makebin :: Integral a => a -> [a] -> String
> makebin n ys = fst (foldr (\x (xs,p) ->
>                   if x > p then ('0':xs,p) else
>                      ('1':xs,p-x)) ("",n) ys )
```

Next, `undo` is a function that takes a list of codes and applies `makebin` to each code.

```
> undo ::  Integral a=>[a] -> [a] -> String
> undo xs ys = foldr (\ x ss-> (makebin x ys)++ss) "" xs
```

```
Main> undo [63,64,65] [1,2,4,8,16,32,64]
"11111100000011000001"
(710 reductions, 1000 cells)
```

The following function chops up the binary string into blocks of size 7 and finds the ASCII code for each block:

```
> ascii :: Integral a=>String -> [a]
> ascii [] = []
> ascii xs = todec ys:ascii zs
>            where (ys,zs) = splitAt 7 xs
```

The function `todec` converts a binary string into a decimal number.

```
> todec :: Integral a => String -> a
> todec = foldl (\s x -> if x=='0' then (2*s) else 1+(2*s)) 0
```

Finally, putting all these functions together gives a definition for `decode`:

```
> decode :: Integral a => [a] -> [a] -> a -> a -> String
> decode xs ys w n =  map chr (filter (>0)
>                        (ascii (undo
>                          (unconvert xs w n) ys)))
```

(The filter is needed to get rid of any zeros occurring doing the coding process.)
Syntax: `decode (code) (s.i. sequence) (multiplier) (mod)`

Using the example in Section 5.2.2:

```
Main> decode [97,190,135,282,135,50,282,148,246,61,153,288,338,241]
[4,9,19,35,79] 23 151
"Dr. Watson"
(7077 reductions, 11543 cells)
```

To construct file handling versions of `decode`, it is useful to define a function `declist` which will be used as a function within `file` and `fileshow`. `read` is used to save writing a list parser (lists are declared as a `Read` type).

```
> declist :: (Integral a,Read a) => [a] -> a -> a -> String -> String
> declist ys x n xs = decode (read xs) ys x n
```

For further details of the file handling versions, see Appendix A.

Now trying some examples:

```
Main> testSup [1,7,12,29,54,107,218,432]
True
(146 reductions, 232 cells)

Main> convert [1,7,12,29,54,107,218,432] 137 900
[137,59,744,373,198,259,166,684]
(406 reductions, 721 cells)

Main> knapFF [137,59,744,373,198,259,166,684] "file1" "file1kn"
(322343 reductions, 492836 cells, 2 garbage collections)
```

```
Main> decodeFS [1,7,12,29,54,107,218,432] 137 900 "file1kn"
Pure ......... the affair at the Victory Ball.
(1033307 reductions, 1603225 cells, 7 garbage collections)
```

5.4 Comments on the Knapsack algorithm

This section briefly looks at some of problems both with the Knapsack algorithm and its implementation.

5.4.1 Problems with the implementation

- The implementation requires that the text is converted to and from binary. If the plaintext has length n, then the binary string has size $7n$. It takes a lot of work to create and use the binary strings.
- To be secure, the superincreasing sequence needs to have a length of hundreds of elements - this is difficult to achieve.
- If the key sequence used has length s and the plaintext has length n then the length of the ciphertext is around $7n / s$. This means that for a large plaintext, a large key will be needed to ensure that the ciphertext is not too unmanageable.
- The values in the superincreasing sequence are usually at least double the previous values. This means that the values become large quite quickly.
- Instead of using all ASCII values between 0 and 127, just the values of the required characters could be used. This would therefore reduce the size of the binary strings used.

5.4.2 Security of the Knapsack algorithm

Despite the fact that the general Knapsack (Subset Sum) problem is very difficult to solve, the Knapsack algorithm has been shown to be insecure [Sch, W1]. Using a superincreasing sequence weakens the “hardness” of the Knapsack problem. This means that Knapsack algorithms are generally not used and so other public key algorithms will have to be implemented.

Chapter 6 The RSA System

In this chapter, another Public Key system - RSA - will be considered. Like the Knapsack algorithm, RSA is based on a "hard" mathematical problem.

6.1 The Mathematics of RSA

RSA [Wsh] derives its name from its three originators - Rivest, Shamir and Adleman. The core of the security of the RSA system is the belief that factoring an integer, made by multiplying two large primes, is considered to be "hard". This section gives a description of the RSA systems and looks at some of the mathematics behind the system.

6.1.1 Description of RSA

Before a message can be sent, the receiver needs to do the following things:

- Find two large primes p and q.
- Work out the product $n = pq$.
- Pick an integer d, which is coprime to $(p-1)(q-1)$
- Work out the integer e such that $ed = 1 \pmod{(p-1)(q-1)}$ (by using the Extended Euclidean algorithm outlined in Section 5.3.1)

The **public key** consists of the integers e and n.
The **private key** consists of the integers d and n.

Encryption: A message M is encrypted by performing the calculation:

$$C = M^e \pmod{n}$$

Decryption: The message is recovered by working out:

$$M = C^d \pmod{n}$$

An essential property of RSA is that a message should remain unchanged after encryption and decryption. Showing this property will make use of a result known as **Fermat's Little Theorem** [LN1]:

Suppose p is prime and $1 \le a < p$ then $a^{p-1} = 1 \pmod{p}$

Consider the text M', which is obtained after encrypting M and then performing decryption on the result:

$$M' = C^d \pmod{n} = \left(M^e\right)^d \pmod{n} = M^{ed} \pmod{n}$$

As $ed = 1 \pmod{(p-1)(q-1)}$, then $\exists \lambda$ such that $ed = 1 + \lambda(p-1)(q-1)$

$$M' = M^{1+\lambda(p-1)(q-1)} \pmod{n} = M \cdot M^{\lambda(p-1)(q-1)} \pmod{n}$$

Consider $M' \pmod{p} = M \cdot 1^{\lambda(q-1)} = M \pmod{p}$ [1] By Fermat's Little Theorem

Consider $M' \pmod{q} = M \cdot 1^{\lambda(p-1)} = M \pmod{q}$ [2] By Fermat's Little Theorem

As $n = pq$, putting [1] and [2] together gives $M' = M \pmod{n}$

The security of the RSA system depends on the secrecy of p and q. If p and q are known, then $(p-1)(q-1)$ can be determined and so the decryption key d can be worked out. It will be seen later that factoring a large number is difficult to perform.

6.1.2 Exponentiation

When using RSA, calculations of expressions of the form M^e (mod n) and C^d (mod n) are needed and it is likely that d or e will be very large. It is therefore necessary to find an efficient way of performing exponentiations reasonably quickly. Just using the exponentiation operator (^) will not always work. Instead a program which works out exponentiation more quickly can be developed - this program exploits the fact that all of the exponentiations will be performed using a modulus

```
> power :: Integral a=> a -> a -> a -> a
> power x p m = loop 1 x p m

> loop :: Integral a => a -> a -> a -> a -> a
> loop y z p m
>       | p == 0            = y
>       | even p            = loop y (rem (z*z) m) (quot p 2) m
>       | otherwise         = loop (rem (y*z) m) z (p-1) m
```

`power x p m` works out x^p (mod m).

The function `loop` splits up the exponent p into powers of 2. The variable z works out the different powers of 2 and y accumulates the necessary powers of 2. For example, x^{25} is calculated by $x^1 \times x^8 \times x^{16}$.

`power` is only better than ^ when the base or exponent is very large.

This calculation:

```
Main> rem (2^123456) 10000
1936
(531 reductions, 45676 cells)
```

takes much longer to work out than:

```
Main> power 2 123456 10000
1936
(689 reductions, 1422 cells)
```

The Haskell implementation of RSA will use the function `power` rather than ^, as the implementation needs a reliable function, even if it could be less efficient for certain cases.

6.1.3 Primes and Factors

Using RSA requires finding two large primes. The generation of two large primes can require a great deal of calculation. One way to find primes could be to generate a list of primes and choose a candidate from the list. A method of generating primes is called the **Sieve of Eratosthenes** [Brd]. To apply the sieve, first write down a list of consecutive numbers from 2 upwards. Now, cross out all the factors of 2 that are bigger than 2, then cross out all the factors of 3 that are greater 3 and so on.

In Haskell:

```
> primes :: [Integer]
> primes =(map head.iterate sieve) [2..]
```

```
> sieve :: [Integer] -> [Integer]
> sieve (p:xs) = [x|x<- xs, rem x p /= 0]
```

For example, to find the 100th prime:

```
Main> primes !! 99
541
(81202 reductions, 169763 cells)
```

The 1000th prime is harder to work out:

```
Main> primes !! 999
(1441042 reductions, 3069473 cells, 72 garbage collections)
ERROR: Garbage collection fails to reclaim sufficient space
```

This method is simple to understand but it consumes a lot of space. Another method, based on sieving, is demonstrated in Appendix C.

6.1.4 Primality testing

If a list of primes cannot be generated quickly then a test could be devised to check whether a number is prime or not. One test could be to see if the number has any factors. The ideal list of factors to try would be the primes. However, as seen in the last section, a list of primes takes a long time to generate.

The following program [P2] first checks whether 2 or 3 is a factor. It then uses the list of numbers 5,7,11,13,17,19,... to check for divisors. The numbers in this list differ alternately by 2 and 4 (the difference is represented by the variable *s*).

```
> factor :: Integer -> (Integer,Integer)
> factor n = factorFrom 2 n 2

> factorFrom m n s
>       |r == 0         = (m,q)
>       |q <= m         = (n,1)
>       |m == 2         = factorFrom 3 n 2
>       |m == 3         = factorFrom 5 n 2
>       |otherwise      = factorFrom (m+s) n (6-s)
>       where (q,r) = quotRem n m
```

This program is fairly quick for numbers with reasonably small factors but it fails to find an answer for large factors.

To check that the 15 digit number 100000000000031 is prime,

```
Main> factor 100000000000031
(100000000000031,1)
(183333396 reductions, 303651140 cells, 1267 garbage collections)
```

takes about 5 minutes.

In general, finding factors of a large number is considered to be a “hard” problem - this is basis for the security of RSA. Therefore, another method to test for primes is needed.

Rabin-Miller Test [LN1]

n is an odd integer. The following algorithm tests whether n is prime.

1. Find q and k such that q is odd and $n = 2^k q + 1$
2. Choose a random integer x such that $2 \le x < n$
3. Work out $y = x^q \pmod n$
4. If $y = 1$ then "True"
5. For $j = 1,2,..,k$
   If $y = n - 1$ then "True"
   Else if $y = 1$ then "False"
   Else let $y = y^2 \bmod n$
6. Then "False"

This algorithm is based on Fermat's Little Theorem (Section 6.1.1). It is possible that the Rabin-Miller algorithm returns "True" when n is not prime. However, if the algorithm is repeated 10 times (and so using different random numbers x), then the chance of an error is less than 1 in a million.

A Haskell implementation of Rabin-Miller will now be developed.

The first task is to find the numbers q and k in Step 1:

```
> step1 :: Integer -> (Integer,Integer)
> step1 n = step1' (n-1) 0

> step1' :: Integer -> Integer -> (Integer,Integer)
> step1' m k
>   |  even m           = step1' (quot m 2) (k+1)
>   |  otherwise        = (m,k)
```

`step1 n` returns (q, k).

Next steps 2 to 4:

```
> test :: Integer -> Integer -> Bool
> test n x
>       | even n        = False
>       | y ==1         = True
>       | otherwise     = testing n y 1
>               where y = power x q n
>                       where q  = fst(step1 n)
```

Step 5:

```
> testing :: Integer -> Integer -> Integer -> Bool
> testing n y j
>       | j > snd(step1 n)      = False
>       | y == n-1              = True
>       | y == 1                = False
>       | otherwise             = testing n (rem  (y*y)  n) (j+1)
```

An example:

```
Main> test 1729 16
True
(347 reductions, 724 cells)
```

But 1729 is composite, as $7 \times 13 \times 19 = 1729$.

However,

```
Main> test 1729 15
False
(707 reductions, 1511 cells)
Main> test 1729 17
False
(582 reductions, 1223 cells)
```

Using the example from earlier and 5 numbers typed in at random:

```
Main> map (test 100000000000031) [6456456,43421,12038347,234724,
86258463]
[True,True,True,True,True]
(10246 reductions, 26855 cells)
```

The test is calculated quite quickly. It is a reasonable test if enough trials are computed.

6.1.5 Choosing the numbers in RSA

To make RSA work easily and securely, the numbers used in the encryption and decryption process have to be chosen carefully.

- p and q need to be "large". Current systems require p and q to be at least a hundred digits long.
- e is usually fairly small. This is so that the encryption process works fairly quickly and also so that d is large - and therefore harder to find.
- If $p - 1$ or $q - 1$ has only small factors then n could be factored more easily.

6.2 RSA using Haskell

The next section shows how a small system could be implemented using Haskell. The Euclidean algorithm function `euc` was outlined in Section 5.3.1 and `power` was described in Section 6.1.2. It is useful to create a function in Haskell that will work out the private key d given e, p and q.

```
> private :: Integral a => a -> a -> a -> a
> private e p q
>       | gcd e pp /= 1         = error "Not coprime"
>       | otherwise             = euc e pp
>       where pp = (p-1)*(q-1)
```

Syntax: `private (public) (prime) (prime)`

6.2.1 Converting the text

The RSA system works on numbers and so it is necessary to convert the plaintext to a sequence of numbers. This will be done by using ASCII values. A block of text will be converted to a single number by supposing that each character acts as a digit in base 128 working from right to left (so the rightmost character is the "unit").

e.g. For the text “AB”, ‘A’ has value 65 and ‘B’ has value 66. So “AB” would have value 66 + (65 × 128) = 8386.

The function `accum` combines a sequence of integers as though they were digits in base 128.

```
> accum :: Integral a=>[a] -> a
> accum = foldl (\a b -> (a*128)+b) 0

Main> accum [65,66]
8386
(50 reductions, 62 cells)
```

`blocks` takes a list of strings and applies `accum` to each element of the list.

```
> blocks :: Integral a=>[String] -> [a]
> blocks [] = []
> blocks (x:xs) = (accum(map (fromInt.ord) x)):(blocks xs)

Main> blocks ["AB","XYZ"]
[8386,1453274]
(142 reductions, 225 cells)
```

6.2.2 Chopping up the text

Before the conversion outlined above can take place, the plaintext needs to be chopped up into blocks.

The size of the blocks is determined by *n* - the product of the primes. For this function to work properly, *n* needs to be bigger than 128. The function `chunk n` actually works out (log *n*)/(log 128).

```
> chunk :: Integral a=>a -> a
> chunk n = chunky n 0 1

> chunky :: Integral a => a -> a -> a -> a
> chunky n p c
>       | n < c         = p-1
>       | otherwise     = chunky n (p+1) (128*c)

Main> chunk 127
0
(63 reductions, 79 cells)
Main> chunk 130
1
(88 reductions, 119 cells)
Main> chunk 39443
2
(113 reductions, 158 cells)
```

The next function `chop` splits the text up into the correct sized chunks:

```
> chop :: Integral b => [a] -> b -> [[a]]
> chop xs m = chop' xs (toInt(chunk m))
```

```
> chop' :: [a] -> Int -> [[a]]
> chop' [] p = []
> chop' xs p = ys: (chop' zs p)
>            where (ys,zs) = splitAt p xs

Main> chop "abcdefgh" 130
["a","b","c","d","e","f","g","h"]
(568 reductions, 888 cells)
Main> chop "abcdefgh" 17000
["ab","cd","ef","gh"]
(513 reductions, 783 cells)
Main> chop "abcdefgh" 3000000
["abc","def","gh"]
(518 reductions, 786 cells)
```

6.2.3 Functional RSA

Now using the results from the previous section, a function that implements RSA can be defined:

```
> rsa :: Integral a=> a -> a -> String -> [a]
> rsa e m ss = map (\x -> power x e m) (blocks(chop ss m))
```

Syntax: `rsa (public key) (product of primes) (string)`

6.2.4 Testing RSA

First pick two “primes”: 1009 and 3511, $1009 \times 3511 = 3542599$.

Using Rabin-Miller and a list of random integers:

```
Main> and(map (test 1009) [45,675,163,300,216,983])
True
(4146 reductions, 8555 cells)
Main> and (map (test 3511) [2540,549,184,91,1642,284])
True
(3646 reductions, 7224 cells)
```

These numbers seem to be prime (in fact as the numbers are small, `factor` could be used to check).

Now, choosing 11 as the public key - work out the private key:

```
Main> private 11 1009 3511
964931
(382 reductions, 668 cells)
```

Using the product and the public key:

```
Main> rsa 11 3542599 "Attack of the Cybermen"
[1645685,186203,2341696,2027503,3286308,3506632,2536082,2376872]
(2809 reductions, 5418 cells)
```

6.3 Decrypting using the public key

This section considers the implementation of Haskell functions to decode an RSA encrypted text.

6.3.1 Rebuilding the text

It is first necessary to define a function which reverses the `accum` function defined in Section 6.2.1.

```
> unaccum :: Integral a => a -> String
> unaccum n = unaccum' n ""

> unaccum' :: Integral a => a -> String -> String
> unaccum' 0 ys = ys
> unaccum' n ys = unaccum' q ((ascii r):ys)
>                 where (q,r) = quotRem n 128

> ascii :: Integral a => a -> Char
> ascii = chr.toInt

Main> unaccum 8386
"AB"
(107 reductions, 156 cells)
```

Next, a function which applies `unaccum` to a list of integers, combining the results into a single string.

```
> undo :: Integral a=> [a] -> String
> undo = foldr (\x xs -> unaccum x ++ xs) []

Main> undo [8386,1453274]
"ABXYZ"
(254 reductions, 386 cells)
```

(Compare to the example for `blocks` in Section 6.2.1.)

6.3.2 A decoding function

Using the functions in the previous section, `decode` can be defined as follows:

```
> decode :: Integral a => [a] -> a -> a -> String
> decode xs d m  = undo (map (\x -> power x d m) xs)
```

Syntax: `decode (coded seq) (private) (product of primes)`

Using the example from Section 6.2.4, the private key was found to be 964931:

```
Main> decode [1645685,186203,2341696,2027503,3286308,3506632,2536082,
2376872] 964931 3542599
"Attack of the Cybermen"
(8316 reductions, 18035 cells)
```

Now, try the file handling versions (see Appendix A) - this time, choose the primes 9887 and 16427 and let the public key be 7. The product is 162413749.

```
Main> private 7 9887 16427
139189231
(278 reductions, 474 cells)

Main> rsaFS 7 162413749 "file2"
[31478925,11961319,9744060,.....,39001933,90437329]
(62620 reductions, 103804 cells)

Main> rsaFF  7 162413749 "file2" "file2rsa"
(62619 reductions, 103807 cells)

Main> decodeFS 139189231 162413749 "file2rsa"
Incurably insomniac ......... throws it away in disgust.
(382663 reductions, 715716 cells, 3 garbage collections)
```

6.4 Limits of the implementation

To ensure that the ciphertext created by the RSA system is secure, it is usually necessary to use primes in the order of at least a hundred digits.

6.4.1 Using large numbers

To test out the implementation, try a modulus at least 1024-bit. The primes chosen are:

$$3^{200}+268 \text{ and } 2^{1024}+643.$$

Using Rabin-Miller and some randomly typed in numbers:

```
Main> and (map (test (2^1024+643)) [948354047,32232834478,
23567236735,325782367,2401785783, 102392356723901])
True
(185602 reductions, 3443624 cells, 15 garbage collections)
Main> and (map (test (3^200+268)) [3252345,2143124,547568756735,
120937384,2932,23032673567,257825782578])
True
(103872 reductions, 716028 cells, 3 garbage collections)
```

So, it seems that the numbers chosen are fairly likely to be prime.

Working out the product:

```
Main> (3^200+268)*(2^1024+643)
47749244432555561900726242552530468536075904054454582734426811054859768640070280980144
75956371853040860932906015547257210579086773998115122921884042937828524939708210034135
29285908225827778957507948998511288453248410729174757480917687018676239189937032982858
56691160895883383746739394983335620611512349584782390316001067217918564131036518530461
96397578271445320085578734479264390927080012759099779988007l
(660 reductions, 1958 cells)
```

To save typing, *n* will be represented by `(3^200+268)*(2^1024+643)` (the actual number could be stored in a file and read in).

Now, choose 5 as the public key and then work out the private key:

```
Main> private 5 (3^200+268) (2^1024+643)
95498488865111123801452485105060937072151808108909165468853622109719537280140561960289
```

```
519127436701278591685657129399283173423930533238707062649219544209710336478698849071703930916228210910976429547559474497947862966828255413867350292418040386559039157841035926532590821214163054276297778432335057120529158026573102659697760535502873207883847195942783638752944813282530274230143404129325176533977753395891
(968 reductions, 3052 cells)
```

Again, to save typing `private 5 (3^200+268) (2^1024+643)` will be used instead of the private key.

Using *file1*, which has around 600 characters.

```
Main> rsaFF 5 ((3^200+268)*(2^1024+643)) "file1" "test1"
(65503 reductions, 143280 cells)
```

This operation took less than a second.

Now, to decode -

```
Main> decodeFS (private 5 (3^200+268) (2^1024+643))
((3^200+268)*(2^1024+643)) "test1"
Pure chance .... Victory Ball.
(456811 reductions, 5997267 cells, 27 garbage collections)
```

This operation took about 20 seconds in total.

For a larger file, the Haskell file *knap.lhs* will be used - it consists of over 5000 characters.

```
Main> rsaFF 5 ((3^200+268)*(2^1024+643)) "knap.lhs" "test2"
(465700 reductions, 1015421 cells, 4 garbage collections)
```

This took about a second.

```
Main> decodeFS (private 5 (3^200+268) (2^1024+643))
((3^200+268)*(2^1024+643)) "test2"
** Test and create superincreasing sequences > testSup :: Integral
a=>[a]-> Bool
(3274719 reductions, 43462498 cells, 243 garbage collections)
```

The whole decryption process took just over 2 minutes - each block of text took about 5 seconds to decode

Generally, the implementation gives a reasonable level of security as it can cope with quite large numbers but the decryption part is slower than encryption.

Chapter 7 Elgamal Encryption

Elgamal encryption is another public key system and, in common with the other systems, it is based on a mathematically "hard" problem.

7.1 The Elgamal algorithm

This section describes the Elgamal algorithm [Sch] and takes a brief look at the underlying mathematics.

7.1.1 Discrete Logarithms

Suppose that a and y are positive real numbers. How easy is it to find x such that:

$$a^x = y \ ?$$

In fact, it is very easy,

$$x = \log_a y = \frac{\log_{10} y}{\log_{10} a}$$

Most scientific calculators have a *log* button which will work this answer out.

However, if a, y and x are all integers and p is a prime, then finding x such that:

$$a^x = y \pmod p$$

is much harder particularly if p is large. This type of problem is known as a **Discrete Logarithm**.

There are many algorithms used to try to solve this type of problem although none can be guaranteed to find a "quick" solution. (*Baby Step/Giant Step* and *Pohlig-Hellman* are two such algorithms [BBS].)

A public key system based on a discrete logarithm problem should be secure if p is large enough and $p - 1$ has a large prime factor.

7.1.2 Developing the system

The receiver needs to decide on a large prime p and a number g ($< p$) - both of these can be shared amongst the other users. The receiver then needs to decide on a private key x and work out:

$$y = g^x \pmod p$$

To **encrypt** a message M, a number k is chosen at random. This k should be coprime to $p - 1$ - this is not essential but it adds to the security of the system. The message C consists of a pair (a, b) where

$$a = g^k \pmod p \text{ and } b = y^k M \pmod p$$

To **decrypt** a plaintext consisting of pairs (a, b), perform the following calculation:

$$M = \frac{b}{a^x} \pmod p = b \times a^{-x} \pmod p = b \times \left(a^{-1}\right)^x \pmod p$$

a^{-1} can be worked out by using the Euclidean Algorithm.

It is necessary to check that a text which has been encrypted and then decrypted remains unchanged.

Suppose that N is the text obtained by encrypting M and then decrypting the result.

$$N = \frac{b}{a^x} (\text{mod } p) = \frac{y^k M}{(g^k)^x} (\text{mod } p) = \frac{y^k M}{(g^x)^k} (\text{mod } p) = \frac{y^k M}{y^k} (\text{mod } p) = M (\text{mod } p)$$

Comments:

- A different random number can be used for each message and therefore for each block of the message. The decryption does not rely on the random number. This adds to the security of the system as the same message can be encrypted differently even using the same values of p, g and x.
- The encryption process generates two parts for the ciphertext - space could be a problem with large plaintexts.

7.1.3 Random numbers

In the encryption part of the Elgamal system, a random number is needed. It is therefore necessary to find a way to generate a list of random numbers. Unfortunately, it is not possible to obtain completely random numbers from a computer. A common way of producing a sequence of "random" numbers (although not cryptographically secure) is by using a relationship of the form:

$r_n = s \times r_{n-1} + t \pmod{n}$, the first value r_0 is usually called the **seed.**

By looking in the C program *random.c* which was used in the second *Operating Systems* [P3] practical, some suitable values can be obtained. In this program, s = 1103515245, t = 12345 and n = 2147483648 = 2^{31}.

A Haskell function can then be specified:

```
> random :: Integral a => a -> [a]
> random s= tail(iterate f s)
>       where f x = rem (1103515245*x + 12345) 2147483648
```

The seed itself also needs to be random. It is possible to use information such as the time of day and how long a user takes to type in a command to generate seeds. However, for simplicity, in the implementation a number will be just be typed in.

7.2 Haskell implementation

This section will show to how to implement a version of Elgamal in Haskell.

7.2.1 Reusing functions

As with RSA, Elgamal works on numbers and so many of the functions developed in the RSA chapter to convert to and from strings can be reused. The list below details the sections in which the relevant definition for each function can be found.

- Section 5.3.1 `euc`
- Section 6.1.2 `power`
- Section 6.2.1 `blocks, chop`
- Section 6.3.1 `undo`

7.2.2 Encryption

The encryption function which deals with text will be developed in stages. The first stage is to produce a function which will take in a list of random numbers and another list of numbers which represents the ciphertext.

```
> elg :: Integral a => a -> a -> [a] -> a -> [a] -> [(a,a)]
> elg g y ks p xs =
>    zip (map (\k -> power g k p) ks)
>        (zipWith (\k m -> rem (m*(power y k p)) p) ks xs)
```

Syntax: `elg g y (random list) p (input list)`

The list of random numbers needs to be at least as long as the length of the input text.

An example:

Pick $p = 719$, $g = 85$ and $x = 290$. Now find y:

```
Main> power 85 290 719
122
(362 reductions, 695 cells)

Main> elg 85 122 [12,20,12] 719 [453,941,529]
[(482,135),(192,361),(482,610)]
(1190 reductions, 2317 cells)
```

If the random list is too small, only some of the plaintext will be encrypted:

```
Main> elg 85 122 [12,20] 719 [453,941,529]
[(482,135),(192,361)]
(822 reductions, 1599 cells)
```

Now, a function is needed that makes use of `random` defined in Section 7.1.3. `random` will take the place of `ks` in `elg`. All of the elements generated by `random` will need to be coprime to the prime p. The test `cpm` can be used with `filter`.

```
> cpm :: Integral a => a -> a -> Bool
> cpm a b = (gcd a b) == 1

Main> filter (cpm 12) [2..12]
[5,7,11]
(1193 reductions, 1736 cells)
```

A seed is needed for `random` - this will be taken as an argument in the function. The length of the input list will determine the start of the random list. Note, when using a text string (see later), the length after conversion to a list of integer is used, not the length of the text.

```
> elgamal :: Integral a => a -> a -> a -> a -> [a] -> [(a,a)]
> elgamal g y k p xs = elg g y rand p xs
>       where rand = drop (length xs) (filter (cpm (p-1))
>                          (map (\n -> rem n p) (random k)))
```

Syntax: `elgamal g y (seed) (prime) (message list)`

Using the same values for g, y and p as above:

```
Main> elgamal 85 122 9 719 [10,34,56]
[(342,302),(408,215),(358,406)]
(4209 reductions, 8122 cells)

Main> elgamal 85 122 9 719 [78,56,49]
[(342,630),(408,481),(358,535)]
(4206 reductions, 8118 cells)
```

Notice that the first components of the elements are the same in the two ciphertexts. However, using a longer list changes the values:

```
Main> elgamal 85 122 9 719 [78,56,49,105]
[(408,28),(358,406),(395,428),(530,562)]
(5289 reductions, 10215 cells)
```

Finally, a function which will take a string as the plaintext and apply the Elgamal encryption to it (and as usual, a file handling version can be defined).

```
> elgtext :: Integral a => a -> a -> a -> a -> String -> [(a,a)]
> elgtext g y k p ss =
>       elgamal g y k p (blocks(chop ss p))
```

Syntax: `elgtext g y (seed) (prime) (string)`

An example - choose $p = 17467$, $g = 476$ and $x = 917$.

```
Main> power 476 917 17467
5088
(479 reductions, 987 cells)
```

So, $y = 5088$.

```
Main> elgtext 476 5088 62 17467 "Hello World"
[(4089,14736),(14653,5661),(11169,10345),(5559,3580),(3122,3169),(136
0,10615)]
(14911 reductions, 30252 cells)
```

7.2.3 Decryption

As with the encryption part, the decryption function will be defined initially to work with just integers. The first stage is to take in the list of pairs which make up the ciphertext and convert each pair back to the corresponding integer of the plaintext. The integer is created by using the rule in Section 7.1.2.

```
> decode' :: Integral a => [(a,a)] -> a -> a -> [a]
> decode' ys x p = foldr(\(a,b) xs ->
>                     (rem (b*power(euc a p) x p) p):xs) [] ys
```

Syntax: `decode' (coded pairs) (private) (prime)`

Using the last example in Section 7.2.2 - $p = 17467$ and $x = 917$.

```
Main> decode' [(4089,14736),(14653,5661),(11169,10345),(5559,3580),
(3122,3169),(1360,10615)] 917 17467
[9317,13932,14240,11247,14700,100]
(4663 reductions, 9447 cells)
```

The output list now has to be turned back into a string - this can be achieved by using the function undo.

```
Main> undo [9317,13932,14240,11247,14700,100]
"Hello World"
(568 reductions, 857 cells)
```

This leads to a function which will turn a list back into a string:

```
> decode :: Integral a => [(a,a)] -> a -> a -> String
> decode ys x p = undo (decode' ys x p)
```

Syntax: `decode (coded pairs) (private) (prime)`

```
Main>  decode  [(4089,14736),(14653,5661),(11169,10345),(5559,3580),
(3122,3169),(1360,10615)] 917 17467
"Hello World"
(5149 reductions, 10179 cells)
```

7.3 Additional Testing

This section will produce some more testing with the Elgamal implementation by using the file handling versions.

7.3.1 Tests with files

This follows the pattern of Section 6.4.1 in that the time to encrypt and decrypt large files will be considered. First the relevant numbers need to be chosen.

$$p = 2^{1024} + 643,\ x = 3^{100} + 123,\ g = 4^{75} + 7$$

Now working out *y*:

```
Main> power (4^75+7) (3^100+123) (2^1024+643)
155192139965914370597194870990263624288093832245103268634666391077300
157400416759176717221144011181919594586137356247178023628492845459923
039767181925861124890644349256939607051094352038808248321798161760868
444187694769167845745030125438274009611342411667831482320582030335332
188069994750785174924699826586425
(8006 reductions, 126880 cells)
```

However, `power (4^75+7) (3^100+123) (2^1024+643)` will be used instead of the full value for *y*.

Again, using *file1*.

```
Main> elgFF (4^75+7) (power (4^75+7) (3^100+123) (2^1024+643)) 493
(2^1024+643) "file1" "test1e"
(123892 reductions, 537289 cells, 2 garbage collections)
```

This took about 2 seconds.

```
Main> decodeFS (3^100+123) (2^1024+643) "test1e"
Pure chance..... Victory Ball.
(478800 reductions, 1920168 cells, 8 garbage collections)
```

The decryption took about 3 seconds.

The Elgamal encryption and decryption process seems to be much quicker that RSA However, the Elgamal encryption fails with large files.

Taking the same values for *p*, *x*, *g* and *y* as above and using the Haskell file *knap.lhs*.

```
Main> elgFF (4^75+7) (power (4^75+7) (3^100+123) (2^1024+643)) 629
(2^1024+643) "knap.lhs" "test2e"
(877255 reductions, 3233779 cells, 15 garbage collections)
```

This took about 6 seconds.

```
Main> decodeFS (3^100+123) (2^1024+643) "test2e"
(1078099 reductions, 1634099 cells, 40 garbage collections)
ERROR: Garbage collection fails to reclaim sufficient space
```

This fails due to lack of space.

7.3.2 GHC

The Elgamal implementation using Hugs fails to work with large numbers and files. Another compiler which could be used is *ghc* (Glasgow Haskell Compiler) [W2]. To use ghc, a function `Main` with type `IO()` needs to be defined - in this case `Main` will be defined using the `decodeFS` expression defined above. The file is then compiled and made into an executable program.

To compile and create an executable program called *decode*.

```
% ghc -O2 -o decode decode.lhs
```

Running:

```
% decode
** Test and create superincreasing sequences> testSup :: Integral
```

So, compiling in ghc successfully decodes the file.

Chapter 8 Conclusions

In the previous chapters, various implementations have been developed. This last chapter looks at some general issues and summarises some conclusions.

8.1 Creating efficient functions

Some of the reasons, outlined in Section 1.2.4, for using functional programming were that programs could be built up by using one function at a time and efficiency can be achieved by looking at these individual functions. To demonstrate this, the original functions used for Section 5.2.2 are shown and the reasons for any changes are given. The original functions are suffixed by "0" to distinguish them from the actual functions used.

The first function developed in Section 5.2.2 was `bin`. The original version was:

```
> bin0 :: Integral a=>a -> String
> bin0 0 =""
> bin0 n = (bin0 q) ++ [chr(ord '0'+p)]
>       where   p = rem n 2
>               q = quot n 2
```

This version uses `++` which is often expensive to use. This was used because the binary string needs to be built up from the right. An alternative would be to create the string from left to right and then `reverse` the result - however this would mean that `reverse` would be used for each for letter of the plaintext and this could become quite inefficient. Instead, an extra string was taken in as a parameter (initially the string is empty) and the binary string is accumulated in this extra string. The other slight change made was the use of `quotRem` instead of working out `quot` and `rem` separately.

The next function that was changed was `zeros`. Here is the original version:

```
> zeros0 :: Int -> String -> String
> zeros0 n s = if n>l then zeros0 n ('0':s) else s
>        where l = length s
```

This version uses `length` at every stage. The length of the string increases by one at each stage and the function stops when the list reaches the correct size. However, the number of zeros needed can be determined at the start of the loop (it is $n - s$, where s is the length of input string) and so this can be passed into the function as an extra parameter.

The functions `input` and `combine` both stayed the same, so here is the original version of `coding`:

```
> coding0 :: String -> [Integer] -> [Integer]
> coding0 "" xs = []
> coding0 ss xs = (combine (take n ss) xs): (coding0 (drop n ss) xs)
>        where n = length xs
```

Two changes were made to this function. First, `splitAt` is used rather than using `take` and `drop` separately. Using `splitAt` means that the list `ss` is only traversed once. As the length of the list `xs` is not changed by the function, it can be passed in as a parameter. In fact, in Section 5.2.2, it is worked out in the function `knapsack`.

Changes were made to many other functions in each of the chapters. Combining changes made to functions means that efficient programs could be developed. Other methods of efficiency could be by using theorems such as Fusion [Brd] or by using program transforms [DMS].

8.2 Timings

One of the main reasons that functional programming is not widely used for cryptography is that it is perceived as being "slow". This section will compare the times for the RSA and Elgamal implementations with another cryptographic algorithm, called PGP, that is used on the ECS network.

PGP stands for Pretty Good Privacy - the details are explained in [Sch] - it is based on RSA. To compare the versions, PGP is set up with a 1024-bit key and a key word. The programs are to be used with *file4* (this is so that each program encrypts and decrypts no more than one block of text).

To speed up the RSA and Elgamal functions, ghc is used. Two versions of each algorithm need to be created to handle the encryption and the decryption processes.

For RSA, $e = 5$, $p = 3^{200} + 268$ and $q = 2^{710} + 337$ (so the modulus is approximately 1024 bits) and the programs are called *rsae* and *rsad*. For Elgamal, the values are the same as those in Section 7.3.1 (the programs are called *elge* and *elgd*).

To work out the times of the functions, the UNIX command *timex* is used. Example timings are given below:

	PGP -encryption	*elge*	*rsae*
real	0.15	0.21	0.05
user	0.11	0.17	0.01
sys	0.02	0.02	0.02

	PGP -decryption	*elgd*	*rsad*
real	1.16	0.18	1.46
user	0.31	0.13	0.89
sys	0.07	0.02	0.03

Comments:

- The real time value for PGP decryption is high. This is because a password needs to be typed in before decryption can take place.
- RSA encryption is very quick, however the decryption time is slow. This happens because the encryption process deals with a small power (7) while the decryption has to deal with a much larger power (around 1024 bit).
- Elgamal encryption takes the longest - the calculation is more complex than RSA and involves random numbers. However, Elgamal is the fastest for decryption.
- The time for the RSA decryption algorithm is (approximately) directly proportional to the number of elements in the list which constitutes the ciphertext. This means that RSA becomes very slow for large files.

The times for the Haskell implementations compare fairly well with PGP.

8.3 Revisiting the aims

One of the aims set out in Chapter 1 was to examine some cryptographic techniques. The algorithms implemented can be split into two main areas - private key and public key. The two private key algorithms were found to be insecure since functions could be created which could crack ciphertexts encrypted using these algorithms. The public key algorithms which were used relied on "hard" mathematical problems. All of the algorithms were quite easy to implement. Splitting up each algorithm into mathematically self-contained parts and writing a function for each part, achieved the implementation. The security of each algorithm was discussed, including the size of numbers needed for the key.

Another aim of the project was to consider an application using the functional style of programming. One of the most widely used features of functional programming is the list. All of the implementations used lists and techniques such as accumulating parameters and using tuples were employed to aid efficiency. The programs developed compared reasonably well with standard implementations and they were quite compact in comparison to imperative programs. The functional style did not really hinder the implementations - everything could be achieved by Haskell.

8.4 Further topics

There are a number of ways in which this project could be extended:

- Other public key algorithms could be implemented such as DES [Sch]. A developing area is Elliptic Curve cryptography and it would be interesting to see how these could be implemented in Haskell [BBS].
- Digital signatures are used to sign and verify documents over the Internet. Functional digital signatures could be created.
- Hugs was the main compiler used for this project - other compilers such as ghc could be tried instead. Also, other functional languages such as ML could be compared with Haskell.
- Efficiency is a major concern for cryptography - can the algorithms be made to run more efficiently?

8.5 Acknowledgements

I would like to thank Richard Brent for guidance on the content and style of the project, particularly with the cryptography parts. I would also like to thank Oege de Moor for his help with ghc and for his comments on the Haskell programs.

Bibliography

Books and Journals

[BBS] Blake, I., Seroussi, G. and Smart, N. (1999). *Elliptic Curves in Cryptography*. Cambridge University Press

[Brd] Bird, R. (1998). *Introduction to Functional Programming Using Haskell (Second Edition)*. Prentice Hall.

[DMS] de Moor, O. and Sittampalam, G. (1998). *Generic Program Transformation*. Oxford University Computing Laboratory.

[Dbn] Durbin, J. R. (1985). *Modern Algebra - An Introduction (2nd Edition)*. Wiley.

[Rcn] Runciman, C. (1997). *Functional Pearl: Lazy wheel sieves and spiral of primes*. Presented in the Journal of Functional Programming 7 (2): 219-225, March 1997, published by Cambridge University press.

[Sch] Schneier, B. (1996). *Applied Cryptography*. Wiley.

[Sgh] Singh, S. (1999). *The Code Book*. Fourth Estate.

[Wsh] Welsh, D. (1988). *Codes and Cryptography*. Oxford University Press.

Websites

[W1] http://www.research.att.com/~amo/doc/crypto.html
This website contains many papers on cryptography. Of particular interest is the paper: *The Rise and Fall of Knapsack Cryptosystems* by A. M. Odlyzko.

[W2] http://www.haskell.org
This website has details about Haskell and compilers such as Hugs and ghc.

Materials from Oxford University Computer Science Courses

[LN1] *Data Structures and Algorithms - Number-theoretic Algorithms and Applications*, Lecture Notes (R. Brent, 2000).

[P1] *Procedural Programming - Laboratory Manual* (M. Spivey, 2000).
In particular, Chapter 3 - Lab 1: Codebreaker.

[P2] *Functional Programming - Practical 1: Sun Factors* (R. Bird, 2000).

[P3] *Operating Systems: Practical Two* (M. Spivey, modified by J. Stoy, 2000)
(The area of interest is the random number generator).

Appendix A Full Listings of Files Created

Chapter 2

caesar.lhs

```
> nlett :: Int -> Char -> Char
> nlett n c
>       | c>='A' && c<='Z' = chr(ord 'A' + (mod (ord c - ord 'A' + n) 26))
>       | otherwise        = c

> capitalise :: Char -> Char
> capitalise c = if isLower c then chr (off + ord c) else c
>       where off = ord 'A' - ord 'a'
> caps :: String -> String
> caps = map capitalise

> isWanted :: Char -> Bool
> isWanted c = (c==' ')||(isUpper c)

> caesar :: Int -> String -> String
> caesar n s = map (nlett n) (filter isWanted (caps s))

> uncaesar :: Int -> String -> String
> uncaesar n  =  caesar (-n)

> freq :: String -> String -> [(Int,Char)]
> freq [] _ = []
> freq (l:ls) xs = (length (filter (==l) xs),l): (freq ls xs)
> analyse :: String -> [(Int,Char)]
> analyse = (freq ['A'..'Z']).caps

> maxfst :: Ord e => (e,f) -> (e,f) -> (e,f)
> maxfst (a,b) (c,d) = if a>=c then (a,b) else (c,d)
> maxpair :: [(Int,a)] -> (Int,a)
> maxpair = foldr1 (maxfst)
> maxlet :: String -> Char
> maxlet = snd.maxpair.analyse

> mostfreq :: [Int]
> mostfreq = map (\s -> ord s - ord 'A') ['E','T','A','I','N','O']

> posslist :: String -> [String]
> posslist ss = map (flip caesar ss) (map (+f) mostfreq)
>       where f = ord 'A'- ord(maxlet ss)

> len :: Int -> [a] -> Bool
> len n ss = (length ss == n)
> sizeword :: Int -> String -> [String]
> sizeword n ss = filter (len n) (words ss)

> doub :: String -> String
> doub [] = []
> doub [x] = []
> doub (x:y:xs) = if (x==y) then x:doub xs else doub (y:xs)

> onelett =["A","I"]
> two1 = ["AM","AN","AS","AT","BE","BY","CO","DO","EG","GO","HE","HI","IE"]
> two2 = ["IN","IS","IT","ME","MR","MY","NO","OF","OH","OK","ON","OP"]
> two3 = ["OR","OX","PI","SO","TO","UP","US","WE"]
> twolett = two1++two2++two3
> doubles = "CDEFGLNOPRST"

> notmember :: Eq a => [a] -> a -> Bool
> notmember [] _     = True
> notmember (x:xs) y = if (x == y) then False else notmember xs y

> errors1 :: String -> Int
> errors1 ss = length (filter (notmember onelett) (sizeword 1 ss))
> errors2 :: String -> Int
> errors2 ss = length (filter (notmember twolett) (sizeword 2 ss))
> errors3 :: String -> Int
> errors3 ss = length (filter (notmember doubles) (doub ss))
> errors :: String -> Int
> errors ss = (errors1 ss) + (errors2 ss) + (errors3 ss)

> minsnd :: Ord f => (e,f) -> (e,f) -> (e,f)
```

Appendix A Full Listings of Files Created

```
> minsnd (a,b) (c,d) = if b <=d then (a,b) else (c,d)
> minpair :: [(a,Int)] -> (a,Int)
> minpair = foldr1 minsnd

> err :: String -> (String,Int)
> err a = (a, errors a)

> decode :: String -> String
> decode = fst.minpair. (map err). (posslist)

> isText :: Char -> Bool
> isText c
>  | ord c < 32         = False
>  | ord c > 127        = False
>  | otherwise          = True

> file :: String -> String -> (String -> String) -> IO()
> file infn outfn func = do xs <- readFile infn
>                           writeFile outfn (filter isText (func xs))
> fileshow :: String -> (String -> String) -> IO ()
> fileshow fn func = do xs <- readFile fn
>                       putStr (filter isText (func xs))

> caesarSF :: Int -> String -> String -> IO()
> caesarSF n ss outfn = writeFile outfn (show(caesar n ss))
> caesarFS :: Int -> String -> IO()
> caesarFS n infn = fileshow infn (show.(caesar n))
> caesarFF :: Int -> String -> String -> IO()
> caesarFF n infn outfn = file infn outfn (caesar n)

> decodeSF :: String -> String -> IO()
> decodeSF ss outfn = writeFile outfn (show(decode ss))
> decodeFF :: String -> String -> IO()
> decodeFF infn outfn = file infn outfn decode
> decodeFS :: String -> IO()
> decodeFS infn = fileshow infn decode

> icaesar :: Int -> IO()
> icaesar n = interact ((caesar n).(takeWhile (/='#')))
```

Chapter 3

vig.lhs

```
> nlett :: Int -> Char -> Char
> nlett n c
>       | c>='A' && c<='Z' = chr(ord 'A' + (mod (ord c - ord 'A' + n) 26))
>       | otherwise        = c

> capitalise :: Char -> Char
> capitalise c = if isLower c then chr (off + ord c) else c
>       where off = ord 'A' - ord 'a'
> cap :: String -> String
> cap = map capitalise

> toval :: Char -> Int
> toval c = ord c - ord 'A'

> vig :: String -> String -> String
> vig k s = vig' (cycle(cap k))  (filter isUpper (cap s))
> vig' :: String -> String  -> String
> vig' _ ""  = ""
> vig' (k:ks) (t:ts) = (nlett (toval k) t) : vig' ks ts

> ivig :: String -> IO()
> ivig ks  = interact ((vig ks).(takeWhile (/='#')))

> revlet :: Char -> Char
> revlet 'A' = 'A'
> revlet c = chr (ord 'B' + ord 'Z' - ord c)
> revkey :: String -> String
> revkey xs = map (revlet) (cap xs)

> vigrev :: String -> String -> String
> vigrev k ss = vig (revkey k) ss
```

Appendix A Full Listings of Files Created

```
> isText :: Char -> Bool
> isText c
>  | ord c < 32         = False
>  | ord c > 127        = False
>  | otherwise          = True

> file infn outfn func = do xs <- readFile infn
>                           writeFile outfn (filter isText (func xs))
> fileshow :: String -> (String -> String) -> IO ()
> fileshow fn func = do xs <- readFile fn
>                       putStr (filter isText (func xs))

> vigSF :: String -> String -> String -> IO()
> vigSF ks ss outfn = writeFile outfn (show(vig ks ss))
> vigFS :: String -> String -> IO()
> vigFS ks infn = fileshow infn (vig ks)
> vigFF :: String -> String -> String -> IO()
> vigFF ks infn outfn = file infn outfn (vig ks)
```

newvig.lhs

```
** Definition for xor **

> twopw :: [Int] -> Int
> twopw = foldr (\ a b -> a + (2*b)) 0

> xor' :: Int -> Int -> [Int]
> xor' 0 0 = []
> xor' a b = if mod a 2 == mod b 2 then 0:xx
>                                  else 1:xx
>                  where xx = xor' (div a 2) (div b 2)
> xor :: Int -> Int -> Int
> xor a b = twopw(xor' a b)
> chars :: Char -> Char -> Char
> chars a b = chr (xor (ord a) (ord b) )

** Vigenere cipher **

> vig' :: String -> String -> String
> vig' _ []          = []
> vig' (t:ts) (s:ss) = (chars t s) : (vig' ts ss)
> vig :: String -> String -> String
> vig ks ss = vig' (cycle ks) ss

** Test for a repeating string **

> rep :: Eq a => [a] -> Int -> Bool
> rep xs n = and (zipWith (==) xs (cycle (take n xs)))
> test :: Int -> [Bool] -> Int
> test _ []    = 0
> test n (x:xs)= if x == True then n else (test (n+1) xs)
> replen :: Eq a => [a] -> Int
> replen xs = test 1 (map (rep xs) [1..(length xs)-2])

** Crib part **

> criblist :: Int -> Int -> String -> String -> [(String,Int,Int)]
> criblist 0 _ _ _       = []
> criblist 0 _ _ []      = []
> criblist k n cs (s:ss) = (vig ss cs, replen (vig ss cs), n-k):
>                                 criblist (k-1) n cs ss

> head1:: [(String,Int,Int)] -> (String,Int,Int)
> head1 [] = ("",-1,-1)
> head1 (x:xs) = x
> snd' :: (a,Int,c) -> Bool
> snd' (a,b,c) = b==0

> crib' :: String -> String -> (String,Int,Int)
> crib' cs ss = head1(dropWhile snd' (criblist k k cs ss))
>              where k = length ss - length cs
> crib :: String -> String -> String
> crib cs ss = if k>0 then take k (drop (k-(mod (p+1) k))
>                                    (cycle (take k xs))) else []
>               where (xs,k,p) = crib' cs ss
```

Appendix A Full Listings of Files Created

```
** Check for matches **

> matches :: String -> String -> Int
> matches [] _          = 0
> matches _ []          = 0
> matches (s:ss) (t:ts) = if s==t then 1 + xx else xx
>                           where xx = matches ss ts

> shifts :: String -> String -> [Int]
> shifts [] ts = []
> shifts ss ts = matches ss ts : shifts (tail ss) ts

> display :: Show a =>[a] ->  Int -> Int -> String
> display  _ _ 0     = []
> display [] _ _     = []
> display (t:ts) n e =
>     ((show n)++":\t"++(show t)++"\n")++display ts (n+1) (e-1)

> analyse' :: String -> String
> analyse' ss = display (shifts (tail ss) ss) 1 25
> analyse :: String -> IO()
> analyse = putStr.analyse'

> isText :: Char -> Bool
> isText c
>  | ord c < 32         = False
>  | ord c > 127        = False
>  | otherwise          = True

> eq :: Int -> String -> String -> [(Char,Int)]
> eq _ [] _          = []
> eq _ _ []          = []
> eq n (s:ss) (t:ts) = if (s == t) then (s,n):xx else xx
>                            where xx = eq (n+1) ss ts
> tries :: Int -> (Char,Int) -> (Char,Int)
> tries k (a,b) = (chars a ' ',mod b k)

> list' :: Int -> String -> String -> [(Char,Int)]
> list' _ [] _  = []
> list' k ss ts = (filter (isText.fst) (map (tries k) (eq 0 ss ts))) ++
>                           (list' k (drop k ss) ts)
> list :: Int -> String -> [(Char,Int)]
> list k ss = list' k (drop k ss) ss

> accum :: [(Char,Int)] -> [(Char,Int,Int)]
> accum [] = []
> accum ((a,b) :ss) = (a,b,c) : accum ss'
>                   where c = 1+(length(filter(==(a,b)) ss))
>                         ss' =  filter (/= (a, b)) ss
> gt3rd :: Ord c => c -> (a,b,c) -> Bool
> gt3rd n (a,b,c) = c>=n
> filter3rd :: Int -> [(Char, Int,Int)] -> [(Char,Int,Int)]
> filter3rd _ [] = []
> filter3rd n ss = filter (gt3rd n) ss

> best :: Int -> Int -> String -> [(Char,Int,Int)]
> best n k ss = filter3rd n (accum (list k ss))
> maxs :: Ord c => (a,b,c) -> (a,b,c) -> (a,b,c)
> maxs (a,b,c) (d,e,f) = if c <f then (d,e,f) else (a,b,c)
> maxlist ::  [(Char,Int,Int)] -> (Char,Int,Int)
> maxlist = foldr (maxs) ('?',0,0)

> first :: (a,b,c) -> a
> first (a,b,c) = a
> third :: (a,b,c) -> c
> third (a,b,c) = c
> eq2nd :: Eq b => b -> (a,b,c) -> Bool
> eq2nd n (a,b,c) = n == b

> find :: Int -> [(Char,Int,Int)] -> Char
> find n ss = (first (maxlist(filter (eq2nd n) ss)))

> key' :: Int -> [(Char,Int,Int)] -> String
> key' 0 _  = []
> key' n ss = find (n-1) ss: (key' (n-1) ss)
> key :: Int -> String -> String
```

```
> key k ss = reverse(key' k  (best 1 k ss))

** File handling part **

> file :: String -> String -> (String -> String) -> IO()
> file infn outfn func = do xs <- readFile infn
>                           writeFile outfn (func xs)
> fileshow :: String -> (String -> String) -> IO ()
> fileshow filename func = do xs <- readFile filename
>                            putStr (func xs)

> vigSF :: String -> String -> String -> IO()
> vigSF ks ss outfn = writeFile outfn (vig ks ss)
> vigFS :: String -> String -> IO()
> vigFS ks infn = fileshow infn (vig ks)
> vigFF :: String -> String -> String -> IO()
> vigFF ks infn out = file infn out (vig ks)

> analyseFS :: String -> IO()
> analyseFS infn = fileshow infn analyse'
> listFS :: Int -> String -> IO()
> listFS k infn = fileshow infn (show.(list k ))
> bestFS :: Int -> Int -> String -> IO()
> bestFS n k infn = fileshow infn (show.(best n k))
> keyFS :: Int -> String -> IO()
> keyFS k infn = fileshow infn (show.(key k))
> cribFS :: String -> String -> IO()
> cribFS cs infn = fileshow infn (show.crib cs)
```

Chapter 5

knap.lhs

```
** Test and create superincreasing sequences

> testSup :: Integral a=>[a]-> Bool
> testSup xs = and(zipWith (>) xs (scanl (+) 0 xs))
> super :: Num a => [a] -> [a]
> super [] = []
> super (x:xs) = sup2 xs [x]
> sup2 :: Num a => [a] -> [a] -> [a]
> sup2 [] y = y
> sup2 (x:xs) y = sup2 xs (y++[1+x+(sum y)])

** Encrypting

> bin' :: Int -> String -> String
> bin' 0 ys = ys
> bin' n ys = bin' q (chr(ord '0' + p) :ys)
>                where (q,p) = quotRem n 2
> bin :: Int -> String
> bin n = bin' n ""

> zeros' :: Int -> String -> String
> zeros' 0 s = s
> zeros' n s = zeros' (n-1) ('0':s)
> zeros :: Int -> String -> String
> zeros n s = if n>l then zeros' (n-l) s else s
>            where l = length s

> input :: String -> String
> input "" =""
> input (x:xs) = (zeros 7(bin (ord x)))++input xs

> combine :: String -> [Integer] -> Integer
> combine "" _           = 0
> combine _ []           = 0
> combine (s:ss) (x:xs) = (kn s x) + (combine ss xs)
> kn :: Num a => Char -> a -> a
> kn '1' n = n
> kn '0' n = 0

> coding :: Int -> String -> [Integer] -> [Integer]
> coding _ "" _  = []
> coding n ss xs = (combine ys xs):(coding n zs xs)
```

Appendix A Full Listings of Files Created

```
>                 where (ys,zs)= splitAt n ss

> knapsack :: [Integer] -> String -> [Integer]
> knapsack xs ss = coding n (input ss) xs
>                  where n = length xs

Format:         knapsack (text string) (sequence)

> convert :: Integral a=>[a] -> a -> a -> [a]
> convert xs w n
>  | gcd w n /= 1    = error "not coprime"
>  | n <= sum xs     = error "modulus too small"
>  | otherwise       = map (\a -> rem (a*w) n) xs

> knapmod :: [Integer] -> Integer -> Integer -> String ->[Integer]
> knapmod xs w n ss= knapsack (convert xs w n) ss

Format:         knapmod (s.i. sequence) (multiplier) (mod) (text)

> iknapsack :: [Integer] -> IO()
> iknapsack ns = interact (show.(knapsack ns).(takeWhile (/='#')))

> isText :: Char -> Bool
> isText c
>  | ord c < 32         = False
>  | ord c > 127        = False
>  | otherwise          = True

> file :: String -> String -> (String -> String) -> IO()
> file infn outfn func = do xs <- readFile infn
>                           writeFile outfn (filter isText (func xs))
> fileshow :: String -> (String -> String) -> IO ()
> fileshow filename func = do xs <- readFile filename
>                            putStr (filter isText (func xs))

> knapSF :: [Integer] -> String -> String -> IO()
> knapSF ns ss outfn = writeFile (outfn) (show(knapsack ns ss))
> knapFS :: [Integer] -> String -> IO()
> knapFS ns infn = fileshow infn (show.(knapsack ns))
> knapFF :: [Integer] -> String -> String -> IO()
> knapFF ns infn outfn = file infn outfn (show.(knapsack ns))

** Euclidean algorithm

> euc :: Integral a => a -> a -> a
> euc a b = mod (euc' a b 0 1) b
> euc' :: Integral a => a -> a -> a -> a -> a
> euc' b r c c1
>  | r == 0    = c1
>  | otherwise = euc' r p (c1-(q*c)) c
>                where (q,p) = quotRem b r

Format:        euc (number) (mod)

> unconvert :: Integral a=>[a] -> a -> a -> [a]
> unconvert  xs w n = map (\a -> rem (a*(euc w n)) n) xs

Format:         unconvert (sequence) (multiplier) (mod)

** Reconstruction the text

> makebin :: Integral a => a -> [a] -> String
> makebin n ys = fst (foldr (\x (xs,p) ->
>                    if x > p then ('0':xs,p) else
>                       ('1':xs,p-x)) ("",n) ys )

> undo ::  Integral a=>[a] -> [a] -> String
> undo xs ys = foldr (\ x ss-> (makebin x ys)++ss) "" xs

> ascii :: Integral a=>String -> [a]
> ascii [] = []
> ascii xs = todec ys:ascii zs
>            where (ys,zs) = splitAt 7 xs

> todec :: Integral a => String -> a
> todec = foldl (\s x -> if x=='0' then (2*s) else 1+(2*s)) 0
```

```
> decode :: Integral a => [a] -> [a] -> a -> a -> String
> decode xs ys w n =  map chr (filter (>0)
>                       (ascii (undo
>                         (unconvert xs w n) ys)))

Format:         decode (code) (s.i. sequence) (multiplier) (mod)

> declist :: (Integral a, Read a) => [a] -> a -> a ->
>                                    String -> String
> declist ys x n xs = decode (read xs) ys x n

> decodeFS :: (Integral a, Read a) => [a] -> a -> a -> String -> IO()
> decodeFS ys x n infn  = fileshow infn (declist ys x n)
> decodeFF :: (Integral a, Read a) => [a] -> a -> a -> String ->
>                                     String -> IO()
> decodeFF ys x n infn outfn =
>                  file infn outfn (declist ys x n)
> decodeSF :: (Integral a) => [a] -> [a] -> a -> a -> String -> IO()
> decodeSF xs ys x n outfn =
>                         writeFile (outfn) (show(decode xs ys x n))
```

Chapter 6

rsa.lhs

```
** Tests for factors **

> factor :: Integer -> (Integer,Integer)
> factor n = factorFrom 2 n 2

> factorFrom m n s
>       |r == 0         = (m,q)
>       |q <= m         = (n,1)
>       |m == 2         = factorFrom 3 n 2
>       |m == 3         = factorFrom 5 n 2
>       |otherwise      = factorFrom (m+s) n (6-s)
>       where (q,r) = quotRem n m

** Rabin-Miller

> step1 :: Integer -> (Integer,Integer)
> step1 n = step1' (n-1) 0
> step1' :: Integer -> Integer -> (Integer,Integer)
> step1' m k
>   |  even m           = step1' (quot m 2) (k+1)
>   |  otherwise        = (m,k)

> test :: Integer -> Integer -> Bool
> test n x
>       | even n        = False
>       | y ==1         = True
>       | otherwise     = testing n y 1
>               where y = power x q n
>                       where q  = fst(step1 n)

> testing :: Integer -> Integer -> Integer -> Bool
> testing n y j
>       | j > snd(step1 n)      = False
>       | y == n-1              = True
>       | y == 1                = False
>       | otherwise             = testing n (rem  (y*y)  n) (j+1)

** Power function

> power :: Integral a=> a -> a -> a -> a
> power x p m = loop 1 x p m

Format:         power (base) (index) (mod)

> loop :: Integral a => a -> a -> a -> a -> a
> loop y z p m
>       | p == 0            = y
>       | even p            = loop y (rem (z*z) m) (quot p 2) m
>       | otherwise         = loop (rem (y*z) m) z (p-1) m
```

Appendix A Full Listings of Files Created

```
> euc :: Integral a => a -> a -> a
> euc a b = mod (euc' a b 0 1) b
> euc' :: Integral a => a -> a -> a -> a -> a
> euc' b r c c1
>  | r == 0    = c1
>  | otherwise = euc' r p (c1-(q*c)) c
>               where (q,p) = quotRem b r

** Works out a public key

> private :: Integral a => a -> a -> a -> a
> private e p q
>       | gcd e pp /= 1         = error "Not coprime"
>       | otherwise             = euc e pp
>       where pp = (p-1)*(q-1)

Format:         private (public) (prime) (prime)

** Encryption

> accum :: Integral a=>[a] -> a
> accum = foldl (\a b -> (a*128)+b) 0
> blocks :: Integral a=>[String] -> [a]
> blocks [] = []
> blocks (x:xs) = (accum(map (fromInt.ord) x)):(blocks xs)

> chunk :: Integral a=>a -> a
> chunk n = chunky n 0 1
> chunky :: Integral a => a -> a -> a -> a
> chunky n p c
>       | n < c         = p-1
>       | otherwise     = chunky n (p+1) (128*c)

> chop :: Integral b => [a] -> b -> [[a]]
> chop xs m = chop' xs (toInt(chunk m))
> chop' :: [a] -> Int -> [[a]]
> chop' [] p = []
> chop' xs p = ys: (chop' zs p)
>            where (ys,zs) = splitAt p xs

> rsa :: Integral a=> a -> a -> String -> [a]
> rsa e m ss = map (\x -> power x e m) (blocks(chop ss m))

Format:         rsa (public) (product of primes) (string)

** Decryption

> unaccum :: Integral a => a -> String
> unaccum n = unaccum' n ""
> unaccum' :: Integral a => a -> String -> String
> unaccum' 0 ys = ys
> unaccum' n ys = unaccum' q ((ascii r):ys)
>                where (q,r) = quotRem n 128

> ascii :: Integral a => a -> Char
> ascii = chr.toInt
> undo :: Integral a=> [a] -> String
> undo = foldr (\x xs -> unaccum x ++ xs) []

> decode :: Integral a => [a] -> a -> a -> String
> decode xs d m  = undo (map (\x -> power x d m) xs)

Format:         decode (coded seq) (private) (product of primes)

> isText :: Char -> Bool
> isText c
>  | ord c < 32         = False
>  | ord c > 127        = False
>  | otherwise          = True

> file :: String -> String -> (String -> String) -> IO()
> file infn outfn func = do xs <- readFile infn
>                           writeFile outfn (filter isText (func xs))
> fileshow :: String -> (String -> String) -> IO ()
> fileshow filename func = do xs <- readFile filename
```

```
>                           putStr (filter isText (func xs))

> rsaSF :: Integral a => a -> a -> String -> String -> IO()
> rsaSF e m ss outfn = writeFile (outfn) (show(rsa e m ss))
> rsaFS :: Integral a => a -> a -> String -> IO()
> rsaFS e m infn = fileshow infn (show.(rsa e m))
> rsaFF :: Integral a => a -> a -> String -> String -> IO()
> rsaFF e m infn outfn = file infn outfn (show.(rsa e m))

> declist :: (Integral a, Read a) => a -> a -> String -> String
> declist d n ss = decode (read ss) d n

> decodeSF :: Integral a => [a] -> a -> a -> String -> IO()
> decodeSF ss d n outfn = writeFile outfn (decode ss d n)
> decodeFS :: (Integral a, Read a) => a -> a -> String -> IO()
> decodeFS d m infn = fileshow infn (declist d m)
> decodeFF :: (Integral a, Read a) =>
>                          a -> a -> String -> String -> IO()
> decodeFF d m infn outfn = file infn outfn (declist d m)

> irsa :: Integral a => a -> a -> IO()
> irsa e m = interact (show.(rsa e m).(takeWhile (/='#')))
```

Chapter 7

elgamal.lhs

```
> power :: Integral a=> a -> a -> a -> a
> power x p m = loop 1 x p m
> loop :: Integral a => a -> a -> a -> a -> a
> loop y z p m
>       | p == 0             = y
>       | even p             = loop y (rem (z*z) m) (quot p 2) m
>       | otherwise          = loop (rem (y*z) m) z (p-1) m

> euc :: Integral a => a -> a -> a
> euc a b = mod (euc' a b 0 1) b
> euc' :: Integral a => a -> a -> a -> a -> a
> euc' b r c c1
>  | r == 0    = c1
>  | otherwise = euc' r p (c1-(q*c)) c
>                where (q,p) = quotRem b r

> random :: Integral a => a -> [a]
> random s = tail(iterate f s)
>       where f x = rem (1103515245*x + 12345) 2147483648

> accum :: Integral a=>[a] -> a
> accum  = foldl (\a b -> (a*128)+b) 0
> blocks :: Integral a=>[String] -> [a]
> blocks [] = []
> blocks (x:xs) = (accum(map (fromInt.ord) x)):(blocks xs)

> chunk :: Integral a=>a -> a
> chunk n = chunky n 0 1
> chunky :: Integral a => a -> a -> a -> a
> chunky n p c
>       | n < c         = p-1
>       | otherwise     = chunky n (p+1) (128*c)

> chop :: Integral b => [a] -> b -> [[a]]
> chop xs m = chop' xs (toInt(chunk m))
> chop' :: [a] -> Int -> [[a]]
> chop' [] p = []
> chop' xs p = ys: (chop' zs p)
>            where (ys,zs) = splitAt p xs

** Encryption **

> elg :: Integral a => a -> a -> [a] -> a -> [a] -> [(a,a)]
> elg g y ks p xs =
>    zip (map (\k -> power g k p) ks)
>        (zipWith (\k m -> rem (m*(power y k p)) p) ks xs)

> elgamal :: Integral a => a -> a -> a -> a -> [a] -> [(a,a)]
```

Appendix A Full Listings of Files Created

```
> elgamal g y k p xs = elg g y rand p xs
>       where rand = drop (length xs) (filter (cpm (p-1))
>                          (map (\n -> rem n p) (random k)))

> cpm :: Integral a => a -> a -> Bool
> cpm a b = (gcd a b) == 1

> elgtext :: Integral a => a -> a -> a -> a -> String -> [(a,a)]
> elgtext g y k p ss =
>       elgamal g y k p (blocks(chop ss p))

> unaccum :: Integral a => a -> String
> unaccum n = unaccum' n ""
> unaccum' :: Integral a => a -> String -> String
> unaccum' 0 ys = ys
> unaccum' n ys = unaccum' q ((ascii r):ys)
>               where (q,r) = quotRem n 128

> ascii :: Integral a => a -> Char
> ascii = chr.toInt
> undo :: Integral a=> [a] -> String
> undo = foldr (\x xs -> unaccum x ++ xs) []

** Decryption **

> decode' :: Integral a => [(a,a)] -> a -> a -> [a]
> decode' ys x p = foldr(\(a,b) xs ->
>                   (rem (b*power(euc a p) x p) p):xs) [] ys

Format:         decode' (coded pairs) (private) (prime)

> decode :: Integral a => [(a,a)] -> a -> a -> String
> decode ys x p = undo (decode' ys x p)

Format:         decode (coded pairs) (private) (prime)

> isText :: Char -> Bool
> isText c
>  | ord c < 32         = False
>  | ord c > 127        = False
>  | otherwise          = True

> file :: String -> String -> (String -> String) -> IO()
> file infn outfn func = do xs <- readFile infn
>                           writeFile outfn (filter isText (func xs))
> fileshow :: String -> (String -> String) -> IO ()
> fileshow filename func = do xs <- readFile filename
>                            putStr (filter isText (func xs))

> elgSF :: Integral a => a -> a -> a -> a -> String -> String -> IO()
> elgSF g y k p ss outfn = writeFile (outfn) (show(elgtext g y k p ss))
> elgFS :: Integral a => a -> a -> a -> a -> String -> IO()
> elgFS g y k p infn = fileshow infn (show.(elgtext g y k p))
> elgFF :: Integral a => a -> a -> a -> a -> String -> String -> IO()
> elgFF g y k p infn outfn = file infn outfn (show.(elgtext g y k p))

> declist :: (Integral a, Read a) => a -> a -> String -> String
> declist x p ss = decode (read ss) x p

> decodeSF :: Integral a => a -> a -> [(a,a)] -> String -> IO()
> decodeSF x p ss outfn = writeFile outfn (decode ss x p)
> decodeFS :: (Integral a, Read a) => a -> a -> String -> IO()
> decodeFS x p infn = fileshow infn (declist x p)
> decodeFF :: (Integral a, Read a) => a -> a -> String -> String -> IO()
> decodeFF x p infn outfn = file infn outfn (declist x p)
```

Appendix B Comparison To Oberon

The variation of the Vigenère cipher and the cracking method discussed in Chapter 3 were based on the Oberon programs created in the first *Procedural Programming* practical. As a comparison, the original Oberon programs are presented here:

crypt.m [used to encrypt a given file]

```
MODULE crypt;

IMPORT Args, Files, Out, Err, Strings, Bit;

CONST
  MaxText = 2048; MaxKey = 64;
VAR
  i, N, K: INTEGER; ch: CHAR;
  text: ARRAY MaxText OF CHAR;
  key, fname: ARRAY MaxKey OF CHAR;
  in: Files.File;

BEGIN
  IF Args.argc < 2 THEN
   Err.String("Usage: crypt key [file]"); Err.Ln;
   HALT(2)
  END;

 Args.GetArg(1, key); K := Strings.Length(key);

  IF Args.argc = 2 THEN in := Files.stdin
   ELSE
    Args.GetArg(2, fname);
    in := Files.Open(fname, "r");
     IF in = NIL THEN
      Err.String("crypt: can't read "); Err.String(fname); Err.Ln;
      HALT(1)
     END
  END;

 N := 0;
  WHILE (N < MaxText) & ~Files.Eof(in) DO
   Files.ReadChar(in, text[N]);
   N := N + 1
  END;

 i:=0;
  WHILE i<N DO
   ch := key[i MOD K];
   text[i]:= CHR(Bit.Xor(ORD(text[i]),ORD(ch)));
   i:=i+1
  END;

 i := 0;
  WHILE i < N DO
   Out.Char(text[i]);
   i := i + 1
  END;
 Out.Ln;

END crypt.
```

crib.m [finds the key by using a crib]

```
MODULE crib;

IMPORT Args, Files, Out, Err, Strings, Bit;

CONST
  MaxText = 2048; MaxKey = 64;
VAR
  i, j, k, l, s, N, K: INTEGER;
  ch: CHAR;found :BOOLEAN;
  crib,text: ARRAY MaxText OF CHAR;
  key, fname: ARRAY MaxKey OF CHAR;
  in: Files.File;

BEGIN
  IF Args.argc < 2 THEN
   Err.String("Usage: crypt key [file]"); Err.Ln;
```

```
   HALT(2)
  END;

 Args.GetArg(1, crib); K := Strings.Length(crib);

  IF Args.argc = 2 THEN in := Files.stdin
  ELSE
   Args.GetArg(2, fname);
   in := Files.Open(fname, "r");
    IF in = NIL THEN
     Err.String("crypt: can't read "); Err.String(fname); Err.Ln;
     HALT(1)
    END
  END;

 N := 0;
  WHILE (N < MaxText) & ~Files.Eof(in) DO
   Files.ReadChar(in, text[N]);
   N := N + 1
  END;

 s:=0; j:=K;found := FALSE;
  WHILE (s + K < N) DO
   i:=0;
    WHILE i < K DO
     ch := crib[i];
     key[i]:= CHR(Bit.Xor(ORD(text[i+s]),ORD(ch)));
     i:=i+1;
    END;

   l:= 0; found:=FALSE;
    WHILE (l < K)&(~found) DO
     k:=0;l:=l+1;
      WHILE (l<= K-2) & (key[k]=key[l+k]) DO
       k:=k+1;
      END;
     found := (l+k=K); j:=K-k;
    END;
   s:=s+1;

    IF found &(j<K)THEN
     l:=0; k:=((s-1) MOD j);
      WHILE (l<j) DO
       i:=((l-k) MOD j);
       Out.Char(key[i]);
       l:=l+1;
      END;
     Out.Ln;
    END;
  END;
END crib.
```

crack1.m [outputs potential key lengths]

```
MODULE crack1;
IMPORT Args, Files, Out, Err;

CONST
  MaxText = 2048; MaxKey = 64;
VAR
  i, s, N: INTEGER;
  count: ARRAY 31 OF INTEGER;
  text: ARRAY MaxText OF CHAR;
  fname: ARRAY MaxKey OF CHAR;
  in: Files.File;

BEGIN
  IF Args.argc <= 1 THEN in := Files.stdin
  ELSE
   Args.GetArg(1, fname);
   in := Files.Open(fname, "r");
    IF in = NIL THEN
     Err.String("crypt: can't read "); Err.String(fname); Err.Ln;
     HALT(1)
    END
  END;

 N := 0;
```

```
  WHILE (N < MaxText) & ~Files.Eof(in) DO
   Files.ReadChar(in, text[N]);
   N := N + 1
  END;

 s:=1;
  WHILE s<=30 DO
   count[s]:=0;
   i:=0;
    WHILE i+s<N DO
      IF text[i]=text[s+i] THEN
       count[s]:=count[s]+1;
      END;
     i:=i+1;
    END;
   Out.Int(s,1);Out.String(": ");Out.Int(count[s],1);Out.Ln;
   s:=s+1;
  END;

END crack1.
```

crack2.m [tries to find the key of a given length]

```
MODULE crack2;

IMPORT Args, Files, Out, Err, Bit, Conv;

CONST
  MaxText = 2048; MaxKey = 64;
VAR
  i,s, N, K, ch: INTEGER;
  text: ARRAY MaxText OF CHAR;
  key, fname: ARRAY MaxKey OF CHAR;
  in: Files.File;

BEGIN
  IF Args.argc < 2 THEN
   Err.String("Usage: crypt key [file]"); Err.Ln;
   HALT(2)
  END;

 Args.GetArg(1, key); K := Conv.IntVal(key);

  IF Args.argc = 2 THEN in := Files.stdin
  ELSE
   Args.GetArg(2, fname);
   in := Files.Open(fname, "r");
    IF in = NIL THEN
     Err.String("crypt: can't read "); Err.String(fname); Err.Ln;
     HALT(1)
    END
  END;

 N:= 0;
  WHILE (N < MaxText) & ~Files.Eof(in) DO
   Files.ReadChar(in, text[N]);
   N := N + 1
  END;

 s:=0;
  WHILE s + K < N DO
   i:=0;
    WHILE i+s < N DO
      IF text[i]=text[i+s] THEN
       ch:= Bit.Xor((ORD(text[i])),32);
        IF (ch > 31) & (ch < 129) THEN
         Out.Int((i MOD K),1);Out.String("   ");Out.Char(CHR(ch));Out.Ln;
        END;
      END;
     i:=i+1;
    END;
   s:=s+K;
  END;

END crack2.
```

Appendix C Finding primes

An alternative to the simple prime sieve outlined in Chapter 6 is given below:

```
> data Wheel = Wheel Int [Int] [Int]

> wheels = Wheel 1 [1] [] : zipWith3 nextSize wheels prime squares

> nextSize (Wheel s ms ns) p q = Wheel (s*p) ms' ns'
>   where
>   (xs,ns') = span (<=q) (foldr (turn 0) (roll (p-1) s) ns)
>   ms' = foldr (turn 0) xs ms
>   roll 0 _ = []
>   roll t o =
>     foldr (turn o) (foldr (turn o) (roll (t-1) (o+s)) ns) ms
>   turn o n rs = let n' = o+n in [n'| mod n' p >0] ++ rs

> prime = spiral wheels prime squares

> spiral (Wheel s ms ns : ws) ps qs = foldr (turn 0) (roll s) ns
>  where roll o = foldr (turn o) (foldr (turn o) (roll (o+s)) ns) ms
>        turn o n rs = let n' = o+n in
>           if n'==2 || n' < head qs then n':rs
>           else dropWhile (<n') (spiral ws (tail ps) (tail qs))

> squares = [p*p | p <- prime]
```

Some tests:

```
Main> prime !! 99
541
(1898 reductions, 2395 cells)
Main> prime !! 999
7919
(416355 reductions, 596270 cells, 2 garbage collections)
```

This method uses less reduction steps and less cells than the versions in Chapter 6.

Full details of how the sieve works is given in [Rcn].

Appendix D Sample Text Files

The file handling versions of the implementation required the use of some example files. Here are the files which were used.

file1 “The Affair of the Victory Ball” by Agatha Christie

```
Pure chance led my friend Hercule Poirot, formerly chief of the
Belgian force, to be connected with the Styles case. His success
brought him notoriety, and he decided to devote himself to the
solving of problems in crime. Having been wounded in the Somme and
invalided out of the Army, I finally took up my quarters with him in
London. Since I have a first-hand knowledge of most of his cases, it
has been suggested to me that I select some of the most interesting
and place them on record. In doing so, I feel that I cannot do better
than begin with that strange tangle which aroused such widespread
public interest at the time. I refer to the affair at the Victory
Ball.
```

file2 “A Void” by Georges Perec (translated by Adair)

```
Incurably insomniac Anton Vowl turns on a light. With a loud and
languorous sigh Vowl sits up, stuffs a pillow at his back, draws his
quilt up around his chin, picks up his whodunit and idly scans a
paragraph or two; but, judging its plot impossibly difficult to
follow in his condition, its vocabulary too whimsically multisyllabic
for comfort, throws it away in disgust.
```

file3 “Remembrance of the Daleks” by Ben Aaronovitch

```
The Doctor puts his hand on Ace's shoulder before they went into the
church. "Time to leave," he said. Ace looked into the Doctor's grey
eyes. "We did good didn't we?" "Perhaps," said the Doctor. "Time will
tell- it always does."
```

file4 A line that uses all 26 letters

```
the quick brown fox jumped over the lazy sleeping dog
```