Lab 1: Codebreaker

by Mike Spivey, adapted by Gavin Lowe, fixes by Joe Pitt-Francis

- This is a practical for *Imperative Programming Part 1*.
- Sign-off deadline is your practical session in Week 4.
- For a mark of S, you must complete the tasks in Sections 3 and 4.
- For a mark of S+, you must also complete the tasks in Section 5.

In this lab, you will write a program for encrypting files that uses a key string to turn a file of ordinary text into a file of seemingly incomprehensible gibberish. The program will also decode the gibberish to give the original file, provided we remember the key. The cipher, the *Vignère Cipher*, is a well-known one, and the baffling appearance of the enciphered text makes it look secure. But in the second part of the lab, we will show that this is far from being true, by implementing not one but two programs that are able to break the cipher within seconds.

The first method of breaking the cipher relies on guessing a word or phrase – a *crib* – that appears in the message: what is called a *known plain-text attack*. "Ah," you say, "that's cheating: when you break a cipher, you're supposed to use just a piece of cipher-text that you've captured." Well, there's some justice in that; but the fact is, many successful attacks on cipher systems have relied on cheating in this way. When the German Enigma cipher was broken during World War II on a daily basis, it was often by monitoring German U-boat signals, and guessing that they would contain the German for phrases like "reconnaissance" and "weather report". Since some German U-boats were under orders to surface first thing each morning and transmit a weather report, this was highly successful. The whole German navy would use the same settings of their cipher machines for a whole day, so once one message (even an unimportant one) had been decoded, the rest would follow easily.

The second method of breaking the cipher needs no crib, but relies on the fact that files of ordinary text contain lots of spaces. The cipher we shall be using has the property that two identical characters that are separated in the plain-text by a multiple of the length of the key will be represented by identical characters in the cipher-text. So we try making two copies of

the cipher-text, shifting one of them by a certain amount, and counting the number of matching characters. When we find a shift that gives a large number of matches, the size of the shift is likely to be a multiple of the key length. This lets us guess the length of the key. Now we look again at those examples where the shift amount is a multiple of the guessed key-length, and bet that a lot of the matches will be due to spaces in the plain text. We can work out what key characters would be needed to explain this; again, we will get some false matches, but usually there will be enough genuine matches to give us a fair chance of guessing the key.

Though it's fun to play with codes and ciphers, this lab has a serious purpose, and that's to make you familiar with using loops and indices to work on arrays of information: in this case, arrays of characters that contain plain-texts, cipher-texts and key strings.

Your lab report should consist of a print-out of your Scala program. The report should contain suitable test data as evidence that your program works: for example, is your solution from Section 4 able to recover a message that has been encrypted with a different key, such as "`PINGPONG`" or "`FALALA`"? Of course, your code should be well commented, and a joy to read!

1 The cipher

I'll explain the cipher first using just letters, as we might do if we were using the cipher by hand to encode messages; then I'll explain a variation that is more suitable for computer implementation.

Suppose we wanted to encrypt the plain-text, "Send guns tomorrow," using the keyword "`BRICKS`". We begin by writing out the message, and writing the keyword underneath, repeating the keyword as often as necessary to match the length of the message:

Plain:	`S E N D G U N S T O M O R R O W`
Key:	`B R I C K S B R I C K S B R I C`
Cipher:	`T V V F Q M O J B Q W G S I W Y`

We now start to add the plain-text and the key letters together, using the convention $A = 0$, $B = 1$, $C = 2$, etc., and reducing the answer modulo 26.[1] So we calculate $S + B = 18 + 1 = 19 = T$, $E + R = 4 + 17 = 21 = V$, and so on. Later in the message, we compute $U + S = 20 + 18 = 38 \equiv 12 = M$, where $\equiv$ denotes reduction modulo 26. All this calculation gives the cipher-text shown in the third line. When the cipher-text is received by someone

[1] That is, if the answer is greater than 25, we subtract 26 to find the equivalent letter.

who knows the key, they can decrypt the original message by reversing the process and subtracting the key letters again.

For computer implementation of this cipher, it's convenient if we can encrypt any file, not just one that consists of upper-case letters. Luckily, there's a standard convention (called the ASCII code) by which all characters are represented by numbers between 0 and 127 ($= 2^7 - 1$); for example, the letter 'A' has code 65. A web search will reveal more information about ASCII codes.

In Scala, given a character `c`, you can obtain the corresponding numeric code using `c.toInt`. Likewise, given an `Int n`, you can obtain the corresponding character using `n.toChar`.[2] Instead of arithmetic modulo 26, we can use arithmetic modulo 128, so that the cipher-text character obtained by encoding the plain-text character `c` using key character `k` is[3]

```
((c.toInt + k.toInt) % 128).toChar
```

We can take a string of text (represented by an array of characters), and encrypt each character in turn in this way.

Actually, there's an even better idea that avoids the need to add the key when encrypting and subtract it when decrypting, and that's to use the operation $\oplus$ of 'bitwise exclusive-or' instead of addition and subtraction.

The bitwise exclusive-or operation takes two binary numbers and combines them bit-by-bit, using the rules $0 \oplus 0 = 1 \oplus 1 = 0$ and $0 \oplus 1 = 1 \oplus 0 = 1$. Thus we would compute $69 \oplus 23$ as follows:

$$69_{10} \oplus 23_{10} = 1000101_2 \oplus 0010111_2 = 1010010_2 = 82_{10}.$$

The operation $\oplus$ is associative and commutative, and has 0 as an identity element. The advantage of using $\oplus$ for encryption instead of addition and subtraction is the cancellation law,

$$(a \oplus b) \oplus b = a,$$

which means that the process of encryption and the process of decryption are identical: we can combine the plain-text character a with the key letter b to obtain the cipher-text character $a \oplus b$; then combine this with b again using the same operation to recover the plain-text character a. This means we need only one program for encryption and decryption instead of two; for this reason, we call the cipher *symmetric*.

[2] These functions don't 'actually do' anything, because characters are always represented by the corresponding numbers inside the machine.

[3] The "mod" operation is written as "`%`" in Scala.

Scala can calculate the bitwise exclusive-or of two `Int`s using the operator `^`. So we can compute the bitwise exclusive-or of two characters using the following function.

```
/** Bit-wise exclusive-or of two characters */
def xor(a:Char, b:Char) : Char = (a.toInt ^ b.toInt).toChar
```

If both characters have codes in the range [0 .. 127] then so will be the value of this expression, so no reduction modulo 128 is needed.

2 Getting started

To get you started, and to prevent you wasting a lot of time with the details of reading and writing files, parsing command-line arguments, etc., you are provided with a template file, `Cipher.scala`, for you to use, available from the course webpage, and shown in Figures 1 and 2.

Figure 1 defines the `xor` function that we've already seen, a function `showCipher` for printing a ciphertext using base 8 or octal (we'll use this below), and functions for reading from file or standard input into an array of characters. It also contains templates for four functions that you will have to implement.[4]

Figure 2 contains just the `main` function, which includes a couple of helper functions; the `main` function parses the command-line arguments and, if everything is in order, calls the appropriate function. For example, once you have written the `encrypt` function, you will be able to use the command

```
scala Cipher -encrypt RUDOLF santa
```

to encrypt the contents of file `santa` using the key "`RUDOLF`", specifying the key and plain-text file as arguments.

At various points, the `main` function has to convert between a `String` and an array of `Char`. If `st` is a string then `st.toArray` produces the corresponding array of `Char`. Conversely, if `a` is an array of `Char`, then `new String(a)` produces the corresponding `String`.

Also on the course webpage are the following files, which you'll need later:

`santa`	One-line sample text
`private1`	Encrypted text
`private2`	Encrypted text

[4] "`???`" is valid Scala, but it will throw a `NotImplementedError` if run.

```scala
object Cipher{
  /** Bit-wise exclusive-or of two characters */
  def xor(a: Char, b: Char) : Char = (a.toInt ^ b.toInt).toChar

  /** Print ciphertext in octal */
  def showCipher(cipher: Array[Char]) =
    for(c <- cipher){ print(c/64); print(c%64/8); print(c%8); print(" ") }

  /** Read file into array */
  def readFile(fname: String) : Array[Char] =
    scala.io.Source.fromFile(fname).toArray

  /** Read from stdin in a similar manner */
  def readStdin() = scala.io.Source.stdin.toArray

  /* ----- Functions below here need to be implemented ----- */

  /** Encrypt plain using key; can also be used for decryption */
  def encrypt(key: Array[Char], plain: Array[Char]) : Array[Char] = ???

  /** Try to decrypt ciphertext, using crib as a crib */
  def tryCrib(crib: Array[Char], ciphertext: Array[Char]) = ???

  /** The first optional statistical test, to guess the length of the key */
  def crackKeyLen(ciphertext: Array[Char]) = ???

  /** The second optional statistical test, to guess characters of the key. */
  def crackKey(klen: Int, ciphertext: Array[Char]) = ???
  ...
}
```

Figure 1: `Cipher.scala`, part 1

```
object Cipher{
  ...
/** The main method just selects which piece of functionality to run */
  def main(args: Array[String]) = {
    // string to print if error occurs
    val errString =
      "Usage: scala Cipher (-encrypt|-decrypt) key [file]\n"+
      "     | scala Cipher -crib crib [file]\n"+
      "     | scala Cipher -crackKeyLen [file]\n"+
      "     | scala Cipher -crackKey len [file]"

    // Get the plaintext, either from the file whose name appears in position
    // pos, or from standard input
    def getPlain(pos: Int) =
      if(args.length==pos+1) readFile(args(pos)) else readStdin

    // Check there are at least n arguments
    def checkNumArgs(n: Int) = if(args.length<n){println(errString); sys.exit}

    // Parse the arguments, and call the appropriate function
    checkNumArgs(1)
    val command = args(0)
    if(command=="-encrypt" || command=="-decrypt"){
      checkNumArgs(2); val key = args(1).toArray; val plain = getPlain(2)
      print(new String (encrypt(key,plain)))
    }
    else if(command=="-crib"){
      checkNumArgs(2); val key = args(1).toArray; val plain = getPlain(2)
      tryCrib(key, plain)
    }
    else if(command=="-crackKeyLen"){
      checkNumArgs(1); val plain = getPlain(1)
      crackKeyLen(plain)
    }
    else if(command=="-crackKey"){
      checkNumArgs(2); val klen = args(1).toInt; val plain = getPlain(2)
      crackKey(klen, plain)
    }
    else println(errString)
  }
}
```

Figure 2: `Cipher.scala`, part 2

3 Encrypting and decrypting

Your first task is to implement the `encrypt` function.

This needs to start by creating an array for the result —call it `cipher`— of the same length as `plain`, i.e. `plain.size`.

Now you need to write a small loop that encrypts the contents of `plain`, putting the result into `cipher`. After this piece of program has run, the first few elements of `cipher` will be

$$t_0 \oplus k_0, t_1 \oplus k_1, \ldots, t_{K-1} \oplus k_{K-1}, t_K \oplus k_0, t_{K+1} \oplus k_1, \ldots,$$

where t_i is the ith character of the original text `plain`, k_i is the ith character of `key`, and K is the length of `key`. Here $a \oplus b$ denotes the bitwise exclusive-or operation on *characters*, as produced by the function `xor`. Notice how the indices used for the key wrap round when we reach the end of the key: you'll need to make provision for that in your program; recall that the mod operator is written as `%` in Scala.

Finally, the `encrypt` function needs to return the value of `cipher`.

Implement this function, using your favourite text editor to edit the file. Compile the file using the command

```
fsc Cipher.scala
```

No doubt you will have made errors that the compiler will detect, and you'll have to decrypt (in another sense) the compiler's messages, edit your program, and compile again. It's quite normal to need several iterations of this process.

Eventually, you will succeed in writing a program that the compiler will accept, and you can try running it, using the command

```
scala Cipher -encrypt RUDOLF santa
```

When you do this, you'll probably find that the answer looks like gibberish. In fact, it will be so garbled that it can't really be displayed properly. Some of the characters that are generated by encryption are from the non-printing part of the ASCII set.[5]

To check the output is right, it would be nice to see it displayed in a more readable form. You can do this by using the standard UNIX program `od` (standing for 'octal dump'), like this:[6]

[5] If you run the program in an `xterm` window, sometimes trying to display the gibberish output can put the window in a funny state where it no longer shows the characters you type. If that happens, the best thing is to shut the window and open a new one.

[6] That's a vertical bar | between `santa` and `od`.

```
scala Cipher -encrypt RUDOLF santa | od -b
```

which produces output like

```
0000 026 060 045 075 154 025 063 073 060 056 140 146 002 071 041 056
0020 077 043 162 067 066 046 042 041 162 070 041 157 055 146 074 060
0040 063 157 056 057 071 060 144 051 043 064 162 026 054 075 045 065
0060 046 070 045 074 140 146 076 072 062 052 154 014 075 075 052 105
0100
```

What has happened here is that the output of the `Cipher` program has been connected to the input of `od`, using the pipe operator "|"; then `od` has displayed the output character-by-character in base 8. Each line of output begins with the offset in the text where the line begins; this is followed by the values in octal of 16 bytes of data. (The function `showCipher` provided in the `Cipher.scala` file produces similar output; you might like to use this function when developing your code.) Thus you can see that the first character of the output has value 026_8, and is one of the non-printing characters of the ASCII set. To see if this is the right character, we need to look at octal versions of the plain-text and the key:

```
od -b santa
```

```
0000 104 145 141 162 040 123 141 156 164 141 054 040 120 154 145 141
0020 163 145 040 142 162 151 156 147 040 155 145 040 141 040 156 145
0040 167 040 142 151 153 145 040 146 157 162 040 103 150 162 151 163
0060 164 155 141 163 054 040 154 157 166 145 040 112 157 150 156 012
0100
```

```
echo -n RUDOLF | od -b
```

```
0000 122 125 104 117 114 106
0006
```

Now we see that the first character of the plain-text is 104_8 (a capital `D`), and the first character of the key is 122_8 (a capital `R`). And

$$104_8 \oplus 122_8 = 01\,000\,100_2 \oplus 01\,010\,010_2 = 00\,010\,110_2 = 026_8,$$

so at least the first character of the output is correct. The great advantage of using octal notation is that it's easy to convert to and from binary: each octal digit corresponds to exactly three binary digits. If the output from your implementation of `encrypt` doesn't match what is shown here, you should check it carefully to find out what the problem might be. In particular, it would be good to check that the key 'wraps round' properly.

Because the cipher is symmetric, you can use the same `encrypt` function to decrypt the message again. If you look at the `main` function in Figure 2, you'll see that it also calls `encrypt` when the `-decrypt` flag is specificed. You can save the cipher-text in a file by redirecting the standard output, like this:

```
scala Cipher -encrypt RUDOLF santa > message
```

or even, perversely

```
scala Cipher -decrypt RUDOLF santa > message
```

The resulting file `message` could be transmitted by insecure e-mail. The recipient (knowing the key) could run `encrypt` again on the contents of `message`:

```
scala Cipher -decrypt RUDOLF message
```

Given the right key, the original message will be recovered.

In fact, the main function also allows the input text to be provided on the command line, so you can run the following

```
scala Cipher -encrypt RUDOLF santa | scala Cipher -decrypt RUDOLF
```

to get your original message back.

You might think that the encrypted version of the message is so garbled that no one could possibly find the message without knowing the key; but despite the off-putting appearance of the cipher-text, this isn't really so, as we'll see very soon.

4 Breaking the cipher using a crib

Suppose you encrypted a file, then forgot the key, but you could remember the first few characters of the file. How could you recover the key? [Hint: you already have a program that will do the job.]

Now suppose you have an encrypted file but not the key, and can only remember or guess a word or phrase that appears *somewhere* in the file – you don't know where. For example, if we managed to tap Santa's phone line, we might expect to capture e-mail messages that contained the word "Christmas" somewhere. So we could use the following plan: take the guessed word, and put it alongside the cipher text at every possible position. For each position, we can work out what the letters of the key-string would have to be for the word to appear in that position of the plain-text.

For the sake of an example, let's go back to the letters-only cipher and addition modulo 26. We put the known plain-text `CHRISTMAS` against the cipher-text at a certain point, and work out what the key letters would have to be.

Cipher: ... W I U Q S W Z M W A L X C I Y S ...
Plain: C H R I S T M A S
Key: O L F R U D O L F

Here, the key letters would have to be `OLFRUDOLF` for the cipher-text to be as shown, because $C + O = 2 + 14 = 16 = Q$, $H + L = 7 + 11 = 18 = S$, and so on. Now we see that for this position, the key letters start to repeat, so it's a fair bet that we have discovered where the guessed word appears in the message, and also what the key letters are. By a bit of arithmetic based on the position of the string in the file, we can work out that the key starts with the `R`, so the key is `RUDOLF`.

If we try other starting positions, we can work out what the key letters would have to be in that case. For example, if we shifted the word `CHRISTMAS` one place further to the right, we would calculate the key letters as follows:

Cipher: ... W I U Q S W Z M W A L X C I Y S ...
Plain: C H R I S T M A S
Key: Q P I E E H Z X K

The key letters must be as shown, because $C + Q = 2 + 16 = 18 = S$, $H + P = 7 + 15 = 22 = W$, and so on. This time, the key letters we've calculated do not start to repeat, so we conclude we have not chosen the right position. Of course, this method relies on having a crib that is a bit longer than the key, and there is always the possibility of getting false matches, where the deduced key sequence starts to repeat, even though the crib does not appear at the chosen position. In that case, the key we guess will not succeed in decrypting the whole message into something sensible. The longer the crib is, the less likely such false matches will be.

If you look at the `main` function in Figure 2, you'll see that if you run the program with the `-crib` flag, it calls the `tryCrib` function, passing in the crib and the text of the file. Your next task is to implement the `tryCrib` function. If the crib really does appear in the plain-text, then the program should print out the key and the plain-text.

The function should repeat steps 1 to 3 below for every starting possible position `start` of the crib in the ciphertext.

1. Work out what the key-characters would have to be for the crib to appear at that position, i.e., find `keyChars` such that

```
text[start..start+K) = crib[0..K) ⊕ keyChars[0..K)
```

where `K` is the length of the crib.

2. Find out whether the computed key-characters contain a repetition of at least two characters: that is, whether there is an index `j` $\leq$ `K-2` such that `keyChars[0..K-j) = keyChars[j..K)` (I suggest you write a separate function that takes `keyChars` and `j` and tests this).

3. If so, then the smallest such value of `j` is the length of the key. Work out from the values of `start` and `j` where in the array `keyChars` the actual key begins, and print it out. Also, work out what the corresponding plain-text is, and print it out.

When you have finished the program and compiled it, you can test the program on the file `message` you prepared in Section 3:

```
scala Cipher -crib Christmas message
```

should give output like

```
RUDOLF
Dear Santa, Please bring me a new bike for Christmas, love John
```

5 A statistical attack (optional)

The method of breaking the cipher that we've used so far depends on knowing or guessing a word or phrase from the plain-text. What if we don't know one and can't guess? In that case, we could try another attack based on the characteristics of text files. Suppose the key is six characters long, and the plain-text contains a word like `bring` that is five characters long.

```
Plain:   Dear Santa, please bring me a new bike for Christmas
Key:     RUDOLFRUDOLFRUDOLFRUDOLFRUDOLFRUDOLFRUDOLFRUDOLFRUDO
Cipher:  ..................X.....X.................X.........
```

Because of the way the key repeats, both spaces will be encrypted with the same key-letter, so they will be represented by the same character in the cipher-text. I've shown that character as an `X` here, although it will be some other character in reality. By chance, the same character will also appear as the encryption of the space between 'for' and 'Christmas', because that space also appears against `R` in the key. Now suppose we take two copies of the cipher-text and shift one of them by six spaces:

Cipher: ..................X.....X..................X.......
Shifted: ..................X.....X..................X....

Two of the `X`'s will line up; let's call that a match of length 6. The text shown has another match of length 6, because there are two spaces separated by the five-letter phrase `a new`, and they are both encrypted with the key-letter `O` to give the same character. Because spaces are common, we can expect any text of reasonable length to show some matches like this. We can also expect matches when we shift the text by a multiple of the key-length. In the example, there are no matches of length 12, but there is at least one match of length 18 and another of length 24, obtained when one or other of the two `X`'s towards the left are shifted over to align with the third `X` at the right. Other longer matches also exist.

The important point is that these matches due to the same character being encoded the same way will only occur when the shift is a multiple of the key-length. Of course, the same character can also appear twice in the cipher-text without it being caused by the same character from the plain-text lining up with the same character from the key; but these random matches will tend to occur at about the same frequency for every shift. So if we count the number of matches for every possible shift, we will probably find that it is greater when the shift is a multiple of the key-length than when it is not. This gives us a good chance of guessing the length of the key.

Once we know the key-length, we can try to guess its individual characters, again using matches between the cipher-text and a shifted copy. This time, we use shifts that are multiples of our guessed key-length, and assume that most matches are caused by spaces lining up. If that is so, then we can work out for each match what the corresponding key-character must be, and from its position, we can work out the position of the character in the key. Again, false matches will lead to wrong guesses for the characters in the key, but if we are lucky, then there will be many more correct guesses than wrong ones.

Now you are ready to implement two functions that carry out a statistical attack based on the high frequency of space characters in text files. One function, `crackKeyLen`, will find the key-length by counting the number of matches for various shifts, showing the counts for its human operator to assess. When the operator has guessed the key-length, function `crackKey` takes the length and the original cipher-text, and by looking for matches with a length that is a multiple of the key-length, produces a list of guessed key characters. We will process this list of guesses with standard UNIX tools to recover the key itself.

In order to apply these methods, you'll need a cipher-text that's a bit longer than the one-line letter to Santa we've been using so far. You can make your own by grabbing any piece of text and encrypting it with any key you choose, but you are provided the files `private1` and `private2` in case you prefer a cipher-text for which you don't know the key.

The first function, `crackKeyLen`, takes a cipher-text and counts the matching characters for various shifts. I made it try each shift amount from 1 to 30, so as to cover small multiples of any modest key size. For each shift amount `shift`, the program counts the number of values of `i` for which `ciphertext(i) = ciphertext(shift+i)`, and prints the results. For example:[7]

```
scala Cipher -crackKeyLen private1
1: 22
2: 15
3: 6
4: 9
5: 12
6: 74
7: 5
8: 5
9: 13
10: 10
11: 8
12: 103
13: 15
...
```

As you can see in this example, there are many more matches when `s` is a multiple of 6 than otherwise. This suggests that the key length is 6, so we try that. It might also be 12, and we might try that later if 6 doesn't work.

You can test your program on the supplied files `private1` and `private2`; this should give you a clearly-defined lengths for the keys.

The second function `crackKey` takes a key-length `klen` and a cipher-text array and looks for likely key letters. Here's the beginning of its output when I used it on a sample file:

```
scala Cipher -crackKey 6 private1
4 L
5 F
```

[7] You'll get different results with the actual file `private1` supplied.

```
1 Y
2 D
4 @
5 F
...
```

The function `crackKey` tries shifting the ciphertext by multiples of `klen`. For each such shift amount `s`, the program again looks for indices `i` such that `ciphertext(i) = ciphertext(s+i)`. Instead of counting them, however, the program works out what key character would explain the match if the corresponding plain-text character were a space. It also works out which of the `klen` characters of the key would be used in this position, and prints the index and the character. To reduce the amount of output, I suppressed characters that were not printable: that is, characters with codes that did not fall between 32 and 127.

The first line of output records a guess that character no. 4 of the key is `L`, and so on. As we'll see in a minute, some of these guesses are wrong: in fact, the fifth line says that the same key character is `@`, and both guesses can't be right. If we're lucky, there will be more right guesses than wrong ones. I suggest you follow exactly the output format shown here, because it will make it easier to do what I suggest below.

If you try your program on the file `private1` with the key-length you guessed above, then you should get a long list of guesses, some right and some evidently wrong.

The problem now is to collate these guesses and hopefully to reject the wrong ones. What we'd like to do is group together all the guesses for each character of the key; we can do that by sorting the file of guesses. Luckily UNIX provides an excellent sorting program ideally suited to the task. Surprisingly enough, this program is called `sort`.[8] So try a command like:

```
scala Cipher -crackKey 6 private1 | sort -n
```

You will see that the guesses for character 0, for character 1, etc., are grouped together. Actually, it would be nicer to summarize the statistics by counting the votes for each character. The UNIX program `uniq` can help with this: it takes a file that has already been sorted and removes duplicate lines; with the flag `-c`, it also counts how many times each line occurred. So now try the command

```
scala Cipher -crackKey 6 private1 | sort -n | uniq -c
```

[8] Try `man sort` for details.

Here's some of the output I got on my example:

```
 1 0 /
 1 0 6
 1 0 >
 1 0 I
23 0 R
 1 0 [
 1 1 :
 1 1 ;
62 1 U
 1 1 [
 1 1 ]
 1 2 !
 1 2 "
 1 2 %
43 2 D
...
```

Each line has three columns: the number of votes, the character index in the key, and the guess at that character. The final step is to discard the guesses that got, say, two votes or fewer. So try this command:

```
scala Cipher -crackKey 6 private1 | sort -n | uniq -c | awk '$1 > 2'
```

The final filter in this pipeline uses the program `awk`; the expression `$1 > 2` is actually a program written in the `awk` language that asks for each line from the input whose first field is a number greater than 2. Here's the output:

```
23 0 R
62 1 U
43 2 D
50 3 O
43 4 L
71 5 F
```

Cracked! I'll leave you to decode the message `private1`, using the key that you've discovered and to decode `private2` in a similar fashion.