Encryption assignment

Izaak van Dongen

January 18, 2018

Contents

The Vigenère cipher (UEncrypt.pas)

This section will concern the unit UEncrypt.pas, which contains my utility functions for Vigenère encryption. The unit's interface looks like this:

```
unit UEncrypt;

interface

uses SysUtils;

const
    alph_s: string = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
    alpha: set of char = ['A'..'Z', 'a'..'z'];
    upper: set of char = ['A'..'Z'];

function vigenere(pt: ansistring; pass: string): ansistring;
function un_vigenere(pt: ansistring; pass: string): ansistring;
function proper_mod(a, b: integer): integer;
```

Listing 1: UEncrypt interface

The Vigenère cipher is a generalisation of the Caesar cipher. Because of this, in fact if I implement a Vigenère routine, I will also have implement the Caesar shift algorithm. It would also implement the one-time pad, as this is simply a Vigenère cipher with a plaintext-length key. I will write my code to work on strings, as strings are guaranteed to be more easily 'seekable'. This is important as for later programs, I'm planning to do a lot of backwards-and-forwards reading through the ciphertext. Ideally I would have written the routine to work with something like 'an iterable of characters', but I have to make some compromises in Pascal.

A short-lived byproduct of this was that any input files were mysteriously cut off. Inspection with wc showed that each file was cut off to exactly 255 characters. Apparently, in some fpc modes 'string' is aliased to 'shortstring', which can only store so many characters. Because of this, I explicitly use 'ansistring' to represent plaintexts and ciphertexts, for clarity, rather than having to use specific compiler options.

Because the Vigenère algorithm is so fundamentally modular, I also needed to write a proper modulo function, such as in Python (that guarantees a positive result for any input). It looks like this:

```
function proper_mod(a, b: integer): integer;
begin
    proper_mod := a mod b;
    if proper_mod < 0 then
        proper_mod := proper_mod + b;
```

```
end;
```

Listing 2: Proper modulo function

Because this function complies with the strict (useful) definition of modulo, this means the later encryption routines that rely on it can be used with any combination of characters, not just words, without worrying about going out of bounds. Note that almost all of them 'rely' on it as the following function will use it.

I also wrote a function to retrieve the 'alphabetic ordinal' of a letter (ie an integer $k : 0 \leq k < 26$):

```
function alpha_ord(c: char): integer;
begin
    alpha_ord := proper_mod(ord(upcase(c)) - 65, 26);
end;
```

Listing 3: Proper modulo function

Note that the previous two functions both behave in a '0-indexed' kind of way - this is because the Vigenère cipher relies heavily on modulo and 0-indexing (A is considered to have the value 0, for example, and the wrapping is implemented by modulo). Because of this, when I'm working with Pascal's actual (1-indexed) strings, careful consideration is needed to avoid off-by-one errors.

Anyway, here's the moment you've all been waiting for - the Vigenère routine:

```
function vigenere(pt: ansistring; pass: string): ansistring;
var
    i: integer = 0;
    c: char;
begin
    vigenere := '';
    for c in pt do
        if c in alpha then begin
            if c in upper then
                vigenere := vigenere + chr(65 + (alpha_ord(c)
                                        + alpha_ord(pass[i + 1])) mod 26)
            else
                vigenere := vigenere + chr(97 + (alpha_ord(c)
                                        + alpha_ord(pass[i + 1])) mod 26);
            i := (i + 1) mod length(pass);
        end else
            vigenere := vigenere + c;
end;
```

Listing 4: Vigenère algorithm

It works quite simply by iterating over the plaintext and whenever it meets a character, advancing the position in the key, and applying the shift using some modular arithmetic, using a conditional to preserve case, and correcting for the flaw that is 1-indexing.

This function uses its output slot to dynamically accumulate the output, rather than making another variable of the same type as the return type named 'answer' or 'ct' or something. This approach will be used by several other functions.

Now, having written a Vigenère routine, it can in fact be recycled to write a decryption routine, by inverting the key, followed by delegation to the encryption routine. In this case, 'inverting the key' means finding the modular additive inverse of each component.

```
function un_vigenere(pt: ansistring; pass: string): ansistring;
var
    i: integer;
begin
    for i := 1 to length(pass) do
        pass[i] := alph_s[1 + proper_mod(-alpha_ord(pass[i]), 26)];
    un_vigenere := vigenere(pt, pass);
end;
```

Listing 5: Vigenère decryption

Reading files (UFiles.pas)

Having done the more fun part, we now have to briefly divert our attention to file handling. This unit handles all of the file reading, providing an interface to a single simple function:

```
function read_file(var f: textfile): ansistring;
var
    c: char;
begin
    read_file := '';
```

```
    while not eof(f) do begin
        read(f, c);
        read_file := read_file + c;
    end;
end;
```

Listing 6: File-reading routine

Command line interface (PVigenere.pas)

At last! A program that will do something. PVigenere will use the previous routines and interface with the command line to allow them to be used. The program is as follows:

```
{$MODE OBJFPC}

program PVigenere;

uses
    UEncrypt, UFiles, SysUtils;

type
    crypt_func = function(pt: ansistring; pass: string): ansistring;

var
    pass: string;
    arg_i: integer;
    encryptor: crypt_func = @vigenere;
begin
    pass := 'n';
    for arg_i := 1 to paramcount do
        if paramstr(arg_i) = '--decrypt' then
            encryptor := @un_vigenere
        else
            try
                pass := alph_s[1 + proper_mod(strtoint(paramstr(arg_i)), 26)];
            except
                on EConvertError do
                    pass := paramstr(arg_i);
            end;
    write(output, encryptor(read_file(input), pass));
end.
```

Listing 7: Command-line interface for Vigenère routines (PVigenere.pas)

This 'parses' the command-line arguments to determine what to do with what parameters. If it encounters the argument '--decrypt' anywhere, it uses the Vigenère decryption routine rather than the encryption routine, which it does by using a function variable to store the encryption routine, as both the encryptor and decryptor have the same signature.

The way it determines the key is as follows: first, the key is set to the default value of 'n'. This results in a Caesar-shift of 13, or a ROT13 cipher. Then, for each argument that is not '--decrypt', it sets to key to this argument, in some sense. First, it tries to parse the argument as an integer caesar shift. If this fails, the argument is used as a Vigenère key. For anything other than the alphabet this behaviour may not be very obvious, as it uses ASCII - 65 mod 26. However, the function is still well-defined and if the same key is used for decryption, it will successfully decrypt.

Because it must 'attempt' to parse an integer, it uses a try/catch statement. Because of this, the mode OBJFPC must be used.

Note also that this program is configured to work with an output file - although that output files is STDOUT, it still writes to a file streams rather than using the primitive write or writeln with no file parameter. Input is similarly read from STDIN. This is pretty standard for a command line program - if the user wants to read from actual input files, this can be easily done with `$ bin/Vigenere key < input.txt > output.txt`, or alternatively they might use the cat command. Leaving to writing to STDOUT is a lot easier for dynamic testing purposes, and doesn't sacrifice any functionality. It should be fairly obvious that reading from STDIN constitutes reading from a file, but here is also a demonstration of how this could be used with an output file rather than writing directly to the terminal:

```
$ echo "There should be one-- and preferably only one --obvious way to do it." | bin/Vigenere Guido > zen.txt
$ cat zen.txt
Zbmus ybwxzj vm rbk-- uvg dxynhfgvtb ctfg rbk --ijywuoa zoe nw gc on.
$ bin/Vigenere --decrypt Guido < zen.txt
There should be one-- and preferably only one --obvious way to do it.
```

Listing 8: Use of an output file

Compiling and testing

This time I wrote another makefile, but one that works generically with my Pascal naming conventions. It assumes that any some `P(.*)\.pas` will compile to `bin/\1`, and also has a depency on all local units, which take the form `U.*\.pas`.

```
.PHONY: all
all: $(shell echo P*.pas | sed -E "s/P([^.]*)\.pas/bin\/\1/g")

bin/%: P%.pas $(wildcard U*.pas)
  fpc -TLinux -o$@ $<
```

Listing 9: The generic FPC makefile

Note that this in fact uses three different languages to represent a generic file - firstly, the generic makefile `bin/%` and `P%.pas`, secondly the shell globs `U*.pas` and `P*.pas`, and lastly the extended regex sed command `s/P([^.]*)\.pas/bin\/\1/g`. The make generics are needed to represent the instructions to build any specific file. The shell glob is used to find all possible program files in a directory, and the regex is used to scrape the executable names from the program file names. Now, I can run 'make', which builds the 'all' target, which is phony so will always try to build all dependecies, which in this case is just 'bin/Vigenere'.

Anyway, having built my executable 'bin/Vigenere', I could now verify that it worked correctly. The first test string I used was AbCdEfGhIjKlMnOpQrStUvWxYz, as this would be useful to test if shifts were working correctly, and if case was being preserved. Don't worry, I didn't type it out by hand.

```
In [2]: "".join(i.lower() if ind & 1 else i for ind, i in enumerate(string.ascii_uppercase))
Out[2]: 'AbCdEfGhIjKlMnOpQrStUvWxYz'
```

Listing 10: Alphabet

Anyway,

```
$ echo "AbCdEfGhIjKlMnOpQrStUvWxYz" | bin/Vigenere 1
BcDeFgHiJkLmNoPqRsTuVwXyZa
$ echo "AbCdEfGhIjKlMnOpQrStUvWxYz" | bin/Vigenere ab
AcCeEgGiIkKmMoOqQsSuUwWyYa
$ echo "AbCdEfGhIjKlMnOpQrStUvWxYz" | bin/Vigenere 3
DeFgHiJkLmNoPqRsTuVwXyZaBc
$ echo "AbCdEfGhIjKlMnOpQrStUvWxYz" | bin/Vigenere 1 | bin/Vigenere --decrypt 1
AbCdEfGhIjKlMnOpQrStUvWxYz
$ echo "AbCdEfGhIjKlMnOpQrStUvWxYz" | bin/Vigenere ab | bin/Vigenere --decrypt ab
AbCdEfGhIjKlMnOpQrStUvWxYz
```

Listing 11: Testing bin/Vigenere

So far, so good. To really put it to the test, I decided to try it with the works of Shakespeare, on the assumption that all the edge cases would probably be covered somewhere along the way.

```
$ cat ~/programmeren/A453/text/shakespeare.txt | wc -c
5458199
$ time cat ~/programmeren/A453/text/shakespeare.txt | bin/Vigenere "William Shakespeare" | wc -c
5458199
cat ~/programmeren/A453/text/shakespeare.txt  0.00s user 0.02s system 5% cpu 0.399 total
bin/Vigenere "William Shakespeare"  0.84s user 0.07s system 99% cpu 0.920 total
wc -c  0.00s user 0.01s system 1% cpu 0.919 total
$ time diff <(cat ~/programmeren/A453/text/shakespeare.txt |
 >             bin/Vigenere "William Shakespeare" |
 >             bin/Vigenere --decrypt "William Shakespeare")
 >           ~/programmeren/A453/text/shakespeare.txt
diff  ~/programmeren/A453/text/shakespeare.txt  0.00s user 0.02s system 1% cpu 1.792 total
```

Listing 12: Shakespeare

This actually did lead to the discovery a bug: as the works of Shakespeare are longer than the size of an integer, the integer (i) tracking the index in letters would overflow, leading to misalignment of the key and the plaintext. This was fixed by applying a modulus to the index whenever modifying it, rather than only when using it to index. See code listing 4. This has now been fixed, and as you can see, the 'diff' command with the source text and the decrypted plaintext exits cleanly, indicating it has been perfectly replicated.

As this is simultaneously an implementation of the one-time pad, we can use it as such. For now it will have to be more of a proof of concept, as the key is limited to <255 characters, as it's stored in a string and taken from argv.

```
$ echo "the quick brown fox jumps over the lazy dog" | bin/Vigenere $(cat *(.))
xuh xyvfr ferdr sre nhpww bylv gkl pncf hbj
$ echo "the quick brown fox jumps over the lazy dog" | bin/Vigenere $(cat *(.)) | bin/Vigenere --decrypt $(cat *(.))
the quick brown fox jumps over the lazy dog
```

```
$ echo "the quick brown fox jumps over the lazy dog" | bin/Vigenere $(cat /dev/urandom | head -c 100)
jih shkno ugein gyr pwsce kgiu ycu mdbl fzk
```

Listing 13: Using the Vigenère program for a one-time pad

Here I've first used the concatenation of all the files in the directory as a recoverable key, to demonstrate also the decryption, followed by 100 cryptographically secure random bytes from /dev/urandom resulting in an unbreakable (undecryptable) encryption.

I also performed some further systematic tests after determining that program could run successfully:

Command	Output
$ echo Dog | bin/Vigenere b	Eph
$ echo Dog | bin/Vigenere 1	Eph
$ echo Dog | bin/Vigenere abc	Dpi
$ echo Eph | bin/Vigenere –decrypt b	Dog
$ echo xy_z2 | bin/Vigenere 33	ef_g2
$ echo ef_g2 | bin/Vigenere –decrypt 33	xy_z2
$ echo "Izaak van Dongen" | bin/Vigenere Python3.7	Xxthy imu Tdlzlb
$ echo "Xxthy imu Tdlzlb" | bin/Vigenere –decrypt Python3.7	Izaak van Dongen

Each of these is correct behaviour.

Determing keyword length

How, having implemented the easy approach for decryption (where you know the keyword), I decided to write a program that, for the user's convenience, doesn't require them to remember their Vigenère key. The first step is to determine the length of the keyword. This is absolutely trivial using some basic statistics and a tiny amount of computing power. For each hypothesised key length, we can simply consider each sequence of letters that would be encrypted by the same keyword letter - ie for a keyword of length l we get l different sub-texts of the ciphertext, where the ith of these is given by $S_i = \{C_{nl+i} : n \in \mathbb{N} \wedge nl+i < |c|\}$. If the hypothesis is correct, these will each have been individually encrypted with a single letter of the key, so we then calculate the index of coincidence of the distribution of each of these sub-texts. The IOC is the probability of any two letters being the same. This has the very obvious property that if the letters change around, the IOC will not change, so the IOC of a text is invariant under any kind of monoalphabetic substitution cipher. It can be calculated from a distribution d indexed from 1 to 26 as follows:

$$\text{IOC} = \frac{\sum_{i=1}^{26} d_i(d_i - 1)}{T(T-1)}$$
$$\text{where } T = \sum_{i=1}^{26} d_i$$

The IOC is a very strong indicator of natural language/English. The expected IOC of English is around 0.067, whereas for uniformly distributed text the expected distribution is $\frac{1}{26} = 0.0385$.

All of the cracking functions have been written in one unit, 'UAttack.pas'. This unit has the following interface:

```pascal
unit UAttack;

interface

uses UFiles, Math;

type
    norm_dist = array['A'..'Z'] of real;
    dist = array['A'..'Z'] of integer;

const
    eng_dist: norm_dist =
   (0.08167, 0.01492, 0.02782, 0.04253, 0.12702, 0.02228, 0.02015, 0.06094,
    0.06966, 0.00153, 0.00772, 0.04025, 0.02406, 0.06749, 0.07507, 0.01929,
    0.00095, 0.05987, 0.06327, 0.09056, 0.02758, 0.00978, 0.02360, 0.00150,
    0.01974, 0.00074);
    alpha: set of char = ['A'..'Z', 'a'..'z'];

function fitness(pt_dist: norm_dist): real;
function IOC(d: dist): real;
```

```
function get_dist(pt: ansistring): dist;
function get_interval(pt: ansistring; start, interval: integer): ansistring;
function clean(pt: ansistring): ansistring;
function normalise(d: dist): norm_dist;
```

Listing 14: UAttack interface

All of these functions will be covered in due course. Things to note are the 'dist' and 'norm_dist' types, which represent a discrete distribution and a normalised probability distribution respectively (ie 'dist' directly represents letter frequencies whereas 'norm_dist' is normalised so that its sum is 1). 'eng_dist' is the distribution of English, taken from Wikipedia.

Now, the first thing you need to do if you want to perform this analysis by IOC is to actually get our S_i substrings. For this, first we'll want to strip away all characters we aren't interested in. This is what the 'clean' function does:

```
function clean(pt: ansistring): ansistring;
var
    c: char;
begin
    clean := '';
    for c in pt do
        if c in alpha then
            clean := clean + c;
end;
```

Listing 15: Clean function

Now, we can extract the sub-text, calculate its distribution, and calculate its distribution's IOC. This part is where the use of strings rather than file streams becomes really crucial, as we want to separately examine different parts of the strings, repeatedly.

```
function get_interval(pt: ansistring; start, interval: integer): ansistring;
var
    i: integer;
begin
    get_interval := '';
    for i := 0 to (length(pt) - start) div interval do
        get_interval := get_interval + pt[i * interval + start + 1];
end;

function get_dist(pt: ansistring): dist;
var
    c: char;
begin
    for c := 'A' to 'Z' do
        get_dist[c] := 0;
    for c in pt do
        if c in alpha then
            inc(get_dist[upcase(c)]);
end;

function IOC(d: dist): real;
var
    total: integer = 0;
    freq: integer;
begin
    IOC := 0;
    for freq in d do begin
        IOC := IOC + freq * (freq - 1);
        total := total + freq;
    end;
    IOC := IOC / ((total * (total - 1)));
end;
```

Listing 16: The rest of the owl (IOC functions)

The only thing of real note here is that the IOC function only makes one pass, where it accumulates its denominator and the total T value.

Now, I needed to write a program that put these functions to use.

```
program PIOC;

uses UAttack, UFiles, Sysutils, Strutils;

function interval_ioc(pt: ansistring; interval: integer): real;
var
    start: integer;
```

```
begin
    interval_ioc := 0;
    for start := 0 to interval - 1 do
        interval_ioc := interval_ioc + IOC(get_dist(get_interval(pt, start, interval)));
    interval_ioc := interval_ioc / interval;
end;

procedure print_IOC(pt: ansistring; max_interval: integer);
var
    interval: integer;
begin
    for interval := 1 to max_interval do begin
        writeln(format('%2d (%.4f) %s', [interval,
                                        interval_ioc(pt, interval),
                                        dupestring('-',
                                                   trunc(interval_ioc(pt, interval) * 500))]));
    end;
end;

begin
    print_ioc(clean(read_file(input)), 20);
end.
```

Listing 17: Command-line interface to IOC functions (PIOC.pas)

This firstly calculates the average IOC for each of the l sub-texts for a hypothesised key length l, and then displays this as a bar graph of l against IOC. My actual source code uses the character U+2796, or 'HEAVY MINUS SIGN', to produce a smooth bar graph. This uses the C-like 'format' function to ensure that everything is properly aligned (see listing 20).

Determining the keyword

Now that we can assume we know the length of the keyword, we can start to attack the keyword itself. To do this, we will need a little brute force - with very low complexity. The number of possibilities that need to be checked as $26l$, which is effectively constant, as for most ciphertexts l will be <20. Therefore the only real factor will be the length of the ciphertext, in which this algorithm should be roughly linear.

To fully automate this, we will need to produce a fitness metric for some text. A very good metric for this is quadgram probability - for each quadgram in the text, use a lookup table of probabilities for quadgrams to occur in English, and use these to accumulate a score for the text. I've used this previously with simulated annealing to attack ciphers that take permutations of the alphabet as a key (eg substitution, playfair, bifid). However, this is not suitable for this algorithm because this isn't a holistic attack - we are attacking separate subsets of the text. Because of this, we won't actually form any adjacent quadgrams, so we can't use this method.

The best criterion we have is 'how similar does this distribution look to English?'. This can be very effectively represented by getting a probability distribution from the sub-text (this is what all of the 'norm_dist' stuff was about), and then for each letter taking the difference between its probability in standard english, its probability in the text, and squaring it, taking the sum of these squares. The squaring step is useful as it ensures only the magnitude of a difference is considered - ie negative differences are still differences. This could also have been done using the abs function, but squaring has the added advantage of being 'harsher' for larger differences, and less harsh for smaller differences. Using this metric, we can determine each letter of the keyword. The following are functions from 'UAttack.pas':

```
function normalise(d: dist): norm_dist;
var
    total: integer = 0;
    i: integer;
    dist_i: char;
begin
    for i in d do
        total := total + i;
    for dist_i := 'A' to 'Z' do
        normalise[dist_i] := d[dist_i] / total;
end;

function fitness(pt_dist: norm_dist): real;
var
    i: char;
begin
    fitness := 0;
    for i := 'A' to 'Z' do
        fitness := fitness + power(pt_dist[i] - eng_dist[i], 2);
end;
```

Listing 18: Fitness library functions

And here is the calling program:

```
{$MODE OBJFPC}

program PCrack;

uses UAttack, UEncrypt, UFiles, Sysutils;

function likely_caes(pt: ansistring): char;
var
    c, best_c: char;
    f, best_f: real;
begin
    best_f := -999;
    for c := 'A' to 'Z' do begin
        f := -fitness(normalise(get_dist(un_vigenere(pt, c))));
        if f > best_f then begin
            best_f := f;
            best_c := c;
        end;
    end;
    likely_caes := best_c;
end;

function likely_vig(pt: ansistring; keyl: integer): string;
var
    strpd: ansistring;
    start: integer;
begin
    strpd := clean(pt);
    likely_vig := '';
    for start := 0 to keyl - 1 do
        likely_vig := likely_vig + likely_caes(get_interval(strpd, start, keyl));
end;

var
    vg_key: string;
    pt: ansistring;
    keyl: integer;
begin
    if paramcount > 0 then
        try
            keyl := strtoint(paramstr(1));
        except
            on EConvertError do begin
                writeln(stderr, 'Invalid key length ', paramstr(1));
                exit;
            end;
        end
    else begin
        writeln(stderr, 'Key length required; see bin/IOC');
        exit;
    end;
    pt := read_file(input);
    vg_key := likely_vig(pt, keyl);
    writeln(output, 'key is ', vg_key);
    writeln(output, 'resulting pt: ', un_vigenere(pt, vg_key));
end.
```

Listing 19: Keyword-cracking program (PCrack.pas)

This first actually defines a function 'likely_caes' to crack a caesar cipher, determining a single letter. This function is then called from 'likely_vig' on each caesar-encrypted subtext to find each letter of the keyword (this of course also works on a caesar cipher, where the length of the keyword is 1).

This also reads the length of the keyword from command-line arguments, which the user should have determined using the IOC program, using various failsafes.

Cracking a cipher

Now I can put it to use. I will use challenge 4b from the cipher challenge as a demonstration. Cracking it takes literally two commands now, each running in a couple of ms:

```
$ cat ~/programmeren/cipher_tools/src/samples/4b.txt | bin/IOC
 1 (0.0429)------------------------
 2 (0.0429)------------------------
```

```
 3 (0.0428) ------------------------
 4 (0.0428) ------------------------
 5 (0.0427) ------------------------
 6 (0.0428) ------------------------
 7 (0.0427) ------------------------
 8 (0.0427) ------------------------
 9 (0.0429) ------------------------
10 (0.0426) ------------------------
11 (0.0433) ------------------------
12 (0.0426) ------------------------
13 (0.0688) ---------------------------------------
14 (0.0427) ------------------------
15 (0.0427) ------------------------
16 (0.0427) ------------------------
17 (0.0428) ------------------------
18 (0.0431) ------------------------
19 (0.0429) ------------------------
20 (0.0421) ------------------------
$ cat ~/programmeren/cipher_tools/src/samples/4b.txt | bin/Crack 13
key is ARCANAIMPERII
resulting pt: OVER THE YEARS THE HEROIC ROLE OF AGRICOLA AT WATLING STREET...
```

Listing 20: Cracking 4b

Source

All involved files, including this LaTeX document, can be found at `https://github.com/elterminad0r/encryption`. An interested reader may also wish to explore `https://github.com/elterminad0r/cipher_tools`, the repository with all the code I produced for the cipher challenge this year. It includes automated crackers for the Hill cipher, autokey cipher and affine Vigenère ciphers by brute force, a framework to allow a user to make substitutions to a text in aid of the cracking of a generic substitution cipher, and crackers for substitution ciphers, playfair ciphers and bifid ciphers written in C using simulated annealing (including quadgram scoring). It also features a slightly derelict Python script that tries to split punctuation-stripped text into words using a prefix tree.