Due Date: COB Friday Week 5.

A Rotating Caesar Cipher

Background:

Encoding text is used for many reasons (spy communication etc.) and this assignment is related to one of the most simple ciphers – Caesar cipher. This – as the name suggests – dates back to Roman Empire times and basically maps letters to new letters by a (cyclic) shift of the alphabet. So, for example, a shift of 1 would map A to B, B to C……. Z to A. There is actually a unix utility rot13 that implements this by (as the name suggests) shifting 13 places in the alphabet (which is nice because it is self-inverse due to the cyclic nature – a shift to 26 returns every letter to the same letter). YOU ARE NOT ALLOWED TO USE rot13 (as it would make the task too easy!) – you MUST follow the instructions below, very precisely.

Our basic strategy will be to use tr (translate) which maps chars to new chars. Look at 'man tr' – you will see you need to give tr parameters that will look something like in these invocations: (from https://gist.github.com/IQAndreas/030b8e91a8d9a407caa6)

Caesar cipher encoding:

```
echo "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG" | tr '[A-Z]'
'[X-ZA-W]'
# output: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD
# Caesar cipher decoding
echo "QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD" | tr '[X-ZA-W]'
'[A-Z]' # output: THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
```

So our first job is:

Task One (40 marks)

Write a shell script called **create_pattern** (EXACTLY that name) that takes two switches as parameters. **create_pattern –n 2 –u** will output the required pattern for Caesar translation by 2 of the upper case alphabet. ([C-ZA-B] . **create_pattern –n 2** will output the required pattern for Caesar translation by 2 of the lower case alphabet. [c-za-b].

Furthermore, the parameters –n and –u can be given in any order. So **create_pattern –n 2 –u** will give exactly the same output as **create_pattern –u –n 2**

NOTE: you CANNOT use a dumb and tedious solution (a cascade of 26 if then elses or a 26 way switch statement. You must do arithmetic on the alphabet to work out the pattern given the shift parameter).

Task Two (20 marks)

Create a second script called **caesar** that takes the same two switches (and again allows any ordering). This script must read its input from stdin (line by line until eof is met) and each line will be Caesar encoded and written to standard out. It will perform this task by calling the first script to create the pattern and using that pattern in a call to tr. Do not assume the search path to your first script includes the current directory! (that is you should invoke the current script with a path starting: ./).

Task Three (40 marks)

Write a third shell script called **decode** that takes a single parameter (a filename) and will for every Caesar shift 1..25 (shifting 0 isn't very interesting) attempt to use that shift to decode the text in the file called filename – sending output to stdout.

This script may assume the input text is uppercase. The output should look exactly like:

Shift 1
(output)
Shift 2
(output)

etc. Where, of course (output) is replaced with the output for that shift.

Optional Challenge 10 extra marks - (manually marked) (no online tests)

Based on the above write a simple tool called **decipher1** that could be used to decipher text – simply look through all 25 decodings looking for something that looks like English and not gibberish.

As a hint for how to do this.. Unix systems typically have a dictionary file that contains lots of English words. One could write a third script that uses the previous scripts to

attempt to decode with one particular shift, and could then look for the decoded words to see how many are in the dictionary. Very likely most wrong decodings will have no words in the dictionary. The right one will have all or almost all words in the dictionary. Write that script that counts how many words were found in the dictionary and picks the winning translation.

As a further extension (decipher2) you can base your assessment of correct key choice on letter frequencies. Select the frequencies that give you the closest match to English langauge frequencies. How well does this approach work?

Submission Instructions

You are to submit electronically through the school SVN repository

The files expected in your repository are the bash executables:

create_pattern
caesar
decode

For the optional part include two other files decipher1 and decipher2. These will be manually marked later on. You will also be given 18 marks for coding style, layout, and commenting.

Detailed Submisison Instructions

The handin key for this exercise is prac1. The following SVN commands will enable you to make a repository for this assignment. Please note the following:

• Perform these steps in the order written once only!

• Replace aaaaaa, where it appears in the commands, with YOUR student id.

• Some commands are long — they must be typed on one line.

Use the Unix “cd” command to change to the place where you want your exercise directory to be stored, then type these commands:

svn mkdir --parents -m "spc prac start" https://version-control.adelaide.edu.au/svn/a1aaaaaa/2018/s1/spc/prac1

(creates this directory in your svn tree)

svn co https://version-control.adelaide.edu.au/svn/a1aaaaaa/2018/s1/spc/prac1 .

(checks out a working copy in your directory) You can now begin work.

You can add a file to the repository by typing the commands:

svn add NAME-OF-FILE

svn commit -m "REASON-FOR-THE-COMMIT"

where “reason-for-the-commit” should be some brief text that explains why you changed the code since the last commit. Note that you only need to add a file once — after that, SVN will “know” it is in the repository. You are now ready to commence working on the exercise.

Make sure you commit your files frequently, in case you have an accident. The University’s SVN repository is very reliable, and is backed up regularly — your computer probably is not... Regular submission is also a good defence against plagiarism by others, since the submissions are dated. We will test the behaviour of your scripts using an automated tester. The tester is thorough, and will find places where your scripts do not work correctly. If it finds an error, it will offer a (vaguish) hint. To encourage you to test your own work, the tester will not be fully available in the first few days before the exercise deadline.

The websubmission system will award up to 82 marks automatically. We will manually check the code for style and commenting and for code associated with the optional bonus exercise. Note that we reserve the right to deduct marks if your code does anything egregious or games the system to obtain marks.

Note again: all your files must be carefully commented to explain the logic of your code.